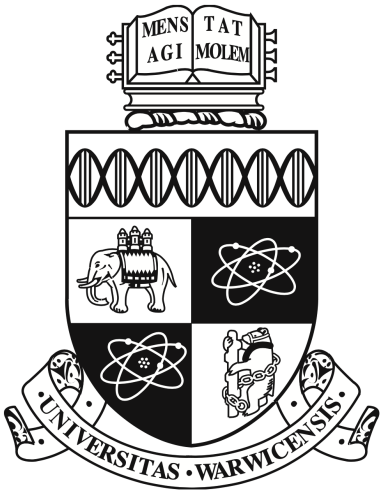


Towards the Use of Mini-Applications in Performance Prediction and Optimisation of Production Codes

by

Andrew Martin Buchanan Owenson

Thesis

Submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

March 2020

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost, I must thank my supervisor, Prof. Stephen Jarvis, for the opportunity to undertake this Ph.D., and for his confidence in me, his enthusiasm, and his guidance and support throughout.

I also must thank past and present members of the High Performance and Scientific Computing group, with special thanks to: Dr. Steven Wright, Dr. Richard Bunt, Dr. James Davis, Prof. Gihan Mudalige, and Dean Chester. Thanks for the mentoring, the discussions, and the prompt support.

This work would not be possible without outside collaboration and sponsorship. I thank the opportunity to collaborate with Rolls-Royce, with special thanks to Dr. Yoon Ho and Matthew Street for their critical support. I also wish to thank Intel for initiating the collaboration, and their technical support throughout.

Lastly, I thank my family for support and encouragement throughout.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- The initial development and validation of MG-CFD as a representative mini-application of HYDRA, described in Chapter 3, was performed by Dr. Richard Bunt. Where this thesis also develops MG-CFD, it is to address deficiencies and improve representativeness.

Parts of this thesis have been previously published by the author in the following:

[44] A. Owenson, S. Wright, R. Bunt, S. Jarvis, Y. Ho, and M. Street. Developing and using a geometric multigrid, unstructured grid mini-application to assess many-core architectures. In 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pages 68–76, 2018

[45] A. Owenson, S. Wright, R. Bunt, Y. Ho, M. Street, and S. Jarvis. An unstructured cfd mini-application for the performance prediction of a production cfd code. Concurrency and Computation: Practice and Experience, 2019

Abstract

Maintaining the performance of large scientific codes is a difficult task. To aid in this task a number of mini-applications have been developed that are more tractable to analyse than large-scale production codes, while retaining the performance characteristics of them. These "mini-apps" also enable faster hardware evaluation, and for sensitive commercial codes allow evaluation of code and system changes outside of access approval processes.

Techniques for validating the representativeness of a mini-application to a target code are ultimately qualitative, requiring the researcher to decide whether the similarity is strong enough for the mini-application to be trusted to provide accurate predictions of the target performance. Little consideration is given to the sensitivity of those predictions to the few differences between the mini-application and its target, how those potentially-minor static differences may lead to each code responding very differently to a change in the computing environment.

An existing mini-application, 'Mini-HYDRA', of a production CFD simulation code is reviewed. Arithmetic differences lead to divergence in intra-node performance scaling, so the developers had removed some arithmetic from Mini-HYDRA, but this breaks the simulation so limits numerical research. This work restores the arithmetic, repeating validation for similar performance scaling, achieving similar intra-node scaling performance whilst neither are memory-bound. MPI strong scaling functionality is also added, achieving very similar multi-node scaling performance.

The arithmetic restoration inevitably leads to different memory-bounds, and also different and varied responses to changes in processor architecture or instruction

set. A performance model is developed that predicts this difference in response, in terms of the arithmetic differences. It is supplemented by a new benchmark that measures the memory-bound of CFD loops. Together, they predict the strong scaling performance of a production 'target' code, with a mean error of 8.8% ($s = 5.2\%$). Finally, the model is used to investigate limited speedup from vectorisation despite not being memory-bound. It identifies that instruction throughput is significantly reduced relative to serial counterparts, independent of data ordering in memory, indicating a bottleneck within the processor core.

Sponsorships and Grants

This research is supported by

1. Rolls-Royce plc.
2. EU Horizon 2020 Clean Sky Project
3. UK Engineering and Physical Sciences Research Council (EPSRC)
4. Intel Corporation

Funding sources are: EP/S005072/1 - Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV); EPSRC Industrial CASE award 15220082

Acronyms

ASiMoV Advanced Simulation and Modelling of Engineering Systems.

AVX advanced vector extensions.

BSP bulk synchronous parallel.

CFD computational fluid dynamics.

CISC complex instruction set computing.

CPI cycles-per-instruction.

DAG directed acyclic graph.

DNN deep neural network.

DSL domain-specific abstraction library.

DT data throughput.

FLOPs floating-point operations per second.

FMA fused multiply-accumulate.

FP floating-point.

FPGA field-programmable gate array.

FU functional unit.

GPGPU general purpose graphic processing unit.

GPU graphic processing unit.

HBM high-bandwidth memory.

HPC high-performance computing.

HT high-throughput.

IDQ instruction decode queue.

ILP instruction-level parallelism.

IMC integrated memory controller.

IP intellectual property.

IPC instructions-per-cycle.

ISA instruction set architecture.

KNL Knights Landing.

LT low-throughput.

MG multigrid.

MIMD multiple instruction, multiple data.

MSR model-specific register.

OSACA Open-Source Architecture Code Analyzer.

PAPI performance application programming interface.

PGAS partitioned global address space.

QAP quadratic assignment problem.

RFO read-for-ownership.

RISC reduced instruction set computing.

RK Runge-Kutta.

SIMD single instruction, multiple data.

SIMT single instruction, multiple thread.

SSE streaming SIMD extensions.

Chapter 1

Introduction

Computational simulation of physical phenomena have become an important component of scientific discovery and engineering design. It can be cheaper and faster than physical experiments, informing and directing early-stage research and design. Simulation can also be necessary when real-world testing is highly challenging, such as assessing aerodynamic performance of a full-scale airframe that is far too large for a wind tunnel [23], or simply illegal. Achieving accurate simulations require the use of a supercomputer, a computing system specialising in numerical computation with performance several magnitudes greater than a typical household or office computer.

One industry quick to adopt computational simulation is aerospace engineering. Initially adopted by national organisations for the design of space and military aircraft, computational fluid dynamics (CFD) soon spread throughout the aerospace industry. CFD has become an essential tool for the historic improvement of airframes and engine turbomachinery, driving improvements to fuel economy and noise pollution [30] [40]. These improvements must continue into the future to further reduce emissions, increasingly demanded by people and Governments, requiring further improvements to the scope and performance of CFD simulations [22].

Early supercomputers consisted of a small number of complex processors specalised at performing floating-point arithmetic, eagerly adopted by national research and defense laboratories such as CERN in Europe and Los Alamos National Laboratory in the United States [12]. This early system design culminated in achieving approximately 1 billion floating-point operations per second (GFLOP/s), but limits of the design and demand for more performance drove a transition to more parallel systems, consisting of a large number of simpler processors executing in parallel (often called a *cluster*). Economics drove a later transition to general-purpose commodity processors. Processor performance steadily improved as transistors

shrunk, increasing operating frequency and density, but this Dennard scaling ended around 2006 as heat and power leakage imposed physical limits - this accelerated the transition into parallelism with multicore CPUs. The one PetaFLOP/s (1 million GFLOPS/s) milestone was passed in 2008, sparking the discussion around how to achieve the next 1000x increase in performance to one ExaFLOP/s. Incorporating projections of expected technology progress, an exascale system would consume 100 MW, an unacceptable amount of power in terms of operating cost and pressure on the local power grid [5]. The exascale challenge emerged with the primary goal of achieving 1 ExaFLOP/s within a 20 MW power cap. This has led to a greater variety of computing architectures, and greater heterogeneity in cluster configurations, in the search for energy efficiency. Particularly, greater adoption of general purpose graphic processing units (GPGPUs) and reduced instruction set computing (RISC) CPU architectures - no supercomputer can meet the exascale challenge without being primarily composed of one of these. More exotic architectures are available offering even greater energy efficiency such as field-programmable gate arrays (FPGAs), but these are difficult to program so remain experimental and areas of research.

The current high-performance computing (HPC) landscape of varied hardware and heterogeneous clusters presents challenges to both cluster operators and users. For operators purchasing an upgrade or new system, they face the complex question of what hardware to select, and in which configuration. They must also factor in the needs of their users, that the change in architecture does not cause unacceptable frustration and prevent important research and simulations. Users must be increasingly prepared to 'port' their software to new architectures and parallelism technologies, meaning to tailor their code execution and data movement to conform well to the architecture design; alternatively, they can select and integrate an appropriate parallel programming abstraction or framework that performs this tailoring behind-the-scenes (such as OpenCL/SYCL, Kokkos, or OP2).

For a simulation application to exploit the performance offered by a modern supercomputer it must be highly parallel, and increasingly incorporate new optimisations and technologies. This is challenging for large legacy codes, as any performance improvement is unknown until after significant time has been invested in software development. Some of these codes face the additional challenge of being proprietary or even classified, impeding the evaluation of potential computing system upgrades – hardware evaluation prior to purchase is integral to system procurement, as application performance depends on more than just the advertised FLOP/s rate. These challenges have spurred the development of 'mini-applications', small codes that capture the key performance characteristics of a target application, either by

containing critical portions of that application or consisting of measurably-similar unrestricted code. This target code can also be referred to as the 'parent' of the mini-application. Where the mini-application shares no code with its parent, but through evaluation is verifed to still capture key characteristics, then it is useful to refer to this instead as a *proxy*-application. The uses of a mini-application derive from the code being much smaller than its parent code, so it is quicker to modify and debug, and easier to compile on new systems. Common use cases are (i) implementing a potential optimisation in code, such as rearranging how data is stored in memory, (ii) evaluating a parallel programming abstraction, such as contrasting partitioned global address space (PGAS) with MPI regarding usability and performance, and (iii) porting to a new processor architecture. The goal of each is to improve the performance of the parent code on current hardware, or to improve the transition of the parent code to new hardware.

1.1 Motivation

Upon the outset of research, the goal was to use an existing proxy-application of a production CFD code, named mini-HYDRA and HYDRA respectively, to address two specific challenges faced by HYDRA. One is to use mini-HYDRA to collect performance data to calibrate an existing performance model of HYDRA, then used to generate performance predictions - this can inform supercomputer scheduling decisions, and accelerate benchmarking of novel hardware. The second challenge is to identify optimisation opportunities with mini-HYDRA that will transfer to HYDRA, allowing it to better utilise the increasing complexity in processor architectures. Mini-HYDRA had been validated using to a published technique that was previously used to validate another unrelated proxy-application, where those original authors then used their validated proxy-application to make predictive assessments of the performance of its target code [59]. However it became clear that while mini-HYDRA had been validated correctly, this technique is ultimately qualitative, as relatively minor static differences between it and the production code can lead to predictive assessments being significantly incorrect, e.g. of predicting no change in performance from a change of instruction set when in fact the target code experiences a significant slowdown.

Given that the proxy-application is more similar than it is different, this poses the question of whether modelling can *transform* proxy-application performance data to the target, allowing the proxy-application to provide accurate quantitative performance prediction.

1.2 Contributions

This thesis makes the following contributions:

- Review mini-HYDRA, an existing proxy-application of the production CFD code HYDRA. Restore arithmetic to correct its CFD, and repeat validation – inevitably some similarity is lost due to different memory-bounds and arithmetic constituents. Add MPI strong scaling, validation shows high similarity in multi-node scaling.

- Presents a new performance projection model for HYDRA, accounting for differences between it and its proxy-application (now called MG-CFD), with which it is possible to project from MG-CFD to HYDRA performance on a range of existing and emerging HPC architectures. This is highly significant for Rolls-Royce plc. as they increase their use of virtual certification and simulation-based engine design;

- Demonstrates that it is possible to use a proxy-application and performance modelling to predict the performance of a production 'target' code, predicting runtime of most expensive loop under strong scaling with a mean error of 8.8% ($s = 5.2\%$) ;

- Combine MG-CFD and the performance model to assess efficacy of auto-vectorising unstructured grid compute. Identifies that vectorised floating-point throughput is approximately half that of serial execution, independent of mesh ordering, indicating a bottleneck within the processor core. Allows expectations of achievable speedup of target application to be determined, not relying on the theoretical maximum of e.g. $4\times$.

1.3 Thesis overview

Chapter 2 covers technical concepts regarding parallel computation and performance analysis. It describes techniques that processor manufacturers have designed to provide parallel computation; modelling techniques for predicting performance of a code; and tools for collecting relevant performance data.

Chapter 3 introduces Rolls-Royce 'HYDRA', a CFD simulation code, detailing aspects of its design that makes HPC challenging. An overview of how HYDRA currently achieves parallel execution is also provided. Finally, an existing proxy-application of HYDRA is summarised, then named mini-HYDRA.

Chapter 4 reviews MG-CFD in detail (formerly named mini-HYDRA), discussing particular challenges faced when designing a proxy-application. Prior modifications made to the CFD to improve a particular aspect of performance similarity are reversed, as they undermine capture of the key computational property influencing performance. This modified code is re-validated, and an alternative solution to challenge is proposed. MG-CFD is further improved with MPI strong scaling, for multi-node performance assessment

Chapter 5 delves deeper into the differences between HYDRA and its proxy-application, and why these lead to difficulty relating performance from the proxy-application to HYDRA loops. A performance model is developed, that seeks to predict the performance difference between the proxy-application and a HYDRA loop, from the known differences in code. Model prediction accuracy is evaluated across several architectures and ISAs. Finally, the model is used to explore the efficacy of vectorising this class of code, identifying that speedup is limited by lower throughput of vectorised FP instructions.

Chapter 6 concludes this thesis with a summary of the work performed, revisiting key arguments and outcomes. Limitations of the work in its current form are discussed, and future work that can address these are detailed.

Chapter 2

Parallel computing and profiling

2.1 Parallel computing

With the end of Dennard scaling, further improvements in computation throughput have been realised through increasing parallelisation. There are several ways by which this can be achieved. Common approachs are instruction-level parallelism (ILP), vector processing, and multi-core processors.

2.1.1 Vector processing

Vector processing, or single instruction, multiple data (SIMD) execution, is the application of the same single operation on multiple data elements. The early vector processors that appeared in supercomputers during the 1970s were essentially data processors, designed to provide a steady stream of data into a scalar but pipelined arithmetic functional unit. The 1974 CDC STAR-100 vector processor streamed data direct from memory to the ALUs, so very wide vectors were needed to mask the latencies and setup costs [18]. The 1976 CRAY-1 introduced vector registers, 64 words wide. A loop of several instructions over 64 words or less can achieve good performance by chaining together a sequence of instructions, storing intermediate results in the registers [51].

The 1990s transition to commodity hardware and massively parallel processor clusters led to vector processing losing their importance, but slowly re-emerged several years later in commodity processors. These were initially 1 word wide in the MMX ISA and limited to integer operations, widening and supporting more data types with successive ISAs: SSE introduced 128-bit FP SIMD, SSE2 added double-precision FP, AVX extended to 256-bit, and AVX-512 extended further to 512-bit. These new SIMD operations differ to the early vector processors in one

fundamental way, they execute one instruction on multiple data elements in lockstep, so as these vectors become wider the demand on memory performance increases proportionally. The data streaming function that was explicitly performed by older vector processors is now performed implicitly by hardware prefetchers that scan for patterns in the memory accesses, and by explicit prefetch instructions inserted into code.

SIMD instruction set architectures

An ISA is the interface between software and functional units within a processor. Being an interface and not an implementation, different processor architectures are free to implement the operations with different circuit designs, which can result in different throughputs of the same instruction. One example of deviation could be the decision of whether to add pipelining to the expensive FP division operation. An ISA is constituted of three parts: the instruction set itself, defining the available operations and their operands; the data types; the registers available for storing state. A SIMD ISA is typically an extension of a base ISA such as x86-64, providing additional operations and registers for vector computation. Recent major SIMD ISAs are SSE4, AVX2, and AVX-512.

1. **SSE4** provides 128-bit double-precision floating-point vector compute, supported by 16× 128-bit vector registers.

2. **AVX2** provides 256-bit versions of SSE4 instructions and registers, and also switches from a two-operand to three-operand format which allows source code to be compiled to fewer instructions.

3. **AVX-512F** extends further to 512-bit, and adds an additional 16 registers for a total of 32.

Unlike previous SIMD ISAs, AVX-512 is composed of several subsets. AVX-512**F** (foundation) and AVX-512**CD** (conflict detection) are universal sets, implemented in all architectures that advertise any AVX-512 support. Then there are several extensions that are specific to certain architectures:

1. Intel Xeon Phi Knights Landing (KNL) implements AVX-512 PF and ER, providing new prefetch instructions and improved approximations of transcendental functions.

2. Intel Xeon processors (Skylake and later) implements AVX-512 DQ, BW and VL, support additional non-floating-point types and execution of AVX-512 instructions on sub-512-bit operands.

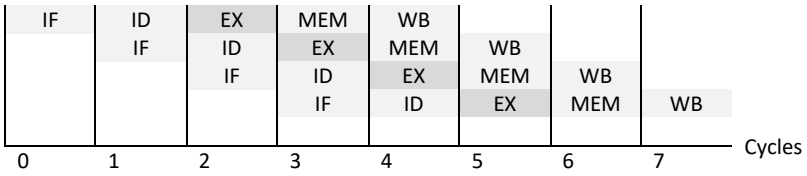


Figure 2.1: 5-stage pipeline of instruction fetch (IF), decode (ID), execute (EX), memory read (MEM), write result (WB). Column of highlighted stages are performed simultaneously.

Thus with AVX-512, a code that is auto-vectorised for different architectures that implement AVX-512F but with different subsets is likely to generate different machine code with differing performance.

2.1.2 Instruction-level parallelism

Fundamentally, all software is compiled to a linear sequence of instructions, generally falling into the categories of data movement, branching, and arithmetic. In the typical fetch-decode-execute instruction cycle, each individual instruction is fetched, decoded, then executed. From early post-war computing efforts several ILP techniques were developed. One is pipelining, which splits the instruction cycle into distinct stages that are processed in a staggered fashion across several clock cycles. Although one individual instruction requires more clock cycles to complete, the processor can process different stages of different instructions simultaneously, increasing overall throughput. This is illustrated in Figure 2.1, which presents the 5-stage pipeline of fetch, decode, execute, memory read, write result, with 5 instructions being processed at different stages simultaneously.

Another ILP technique exists in superscalar processors, which extends pipelining by supporting several parallel instances of each stage. The number of stage instances is a decision of processor architecture design, constrained by the available transistor budget and the need to balance throughput with the other stages. An example is shown in Figure 2.2, which shows for the Xeon Skylake architecture the pipeline portion regarding instruction dispatch to functional units, and memory R/W [28]. The instruction decode queue (IDQ) receives up to 15 decoded μ-ops/cycle from three distinct instruction decoders. μ-ops are a feature of complex instruction set computing (CISC), where assembly-level instructions can be mapped to several transistor-level micro-operations, for example to a memory load μ-op and an arithmetic μ-op. The IDQ then dispatches up to 4 μ-ops/cycle to the scheduler, in turn

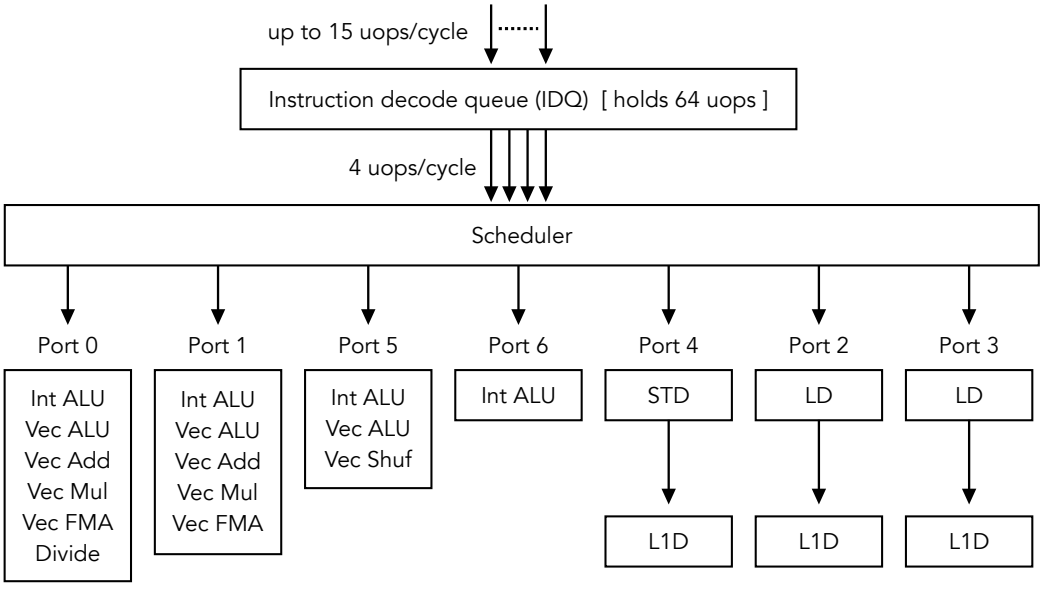


Figure 2.2: Instruction dispatch and issue stages of Xeon Skylake pipeline [28]

dispatching to the appropriate functional units as they become free. The number and distribution of functional units is another architecture design decision, also constrained by the transistor budget and need to provide good performance to many different codes. In this example two FP functional units (FUs) reside across ports 0 and 1, and four integer FUs reside across ports 0, 1, 5 and 6. Thus a code consisting mostly of FP arithmetic and with a high arithmetic intensity will be bottlenecked by these two ports to a maximum of 2 FP operations/cycle, despite the plethora of other functional units and higher throughput of instruction decoding. Memory load and store FUs typically reside on separate ports, in this example on ports 2, 3 and 4. Understanding how a particular code flows through a particular processor pipeline can provide insight into how well it is utilising that architecture, and what architectural changes would improve its throughput.

The first superscalar processor was the Cray CDC 6660 in 1964, and this also introduced a third ILP technique - out-of-order execution. Complementing superscalar execution, this is an instruction scheduling strategy that opportunistically executes a sequence of instructions in a non-linear order to increase utilisation of superscalar resources, reducing overall runtime. The first scheduling algorithm was Thornton's 'scoreboard', that tracks availability of functional units and registers [57]. Data dependencies between instructions are implicitly handled by tracking the occupancy of registers. As resources become available, the scoreboard can dispatch

queued instructions for execution. The Tomasulo algorithm was proposed and implemented several years later, improving the design with reservation stations distributed across the functional units, replacing the single centralised scoreboard [58]. These abstract away registers - an instruction now operates on logical registers, and the station selects whichever physical registers are available. Modern out-of-order execution logic has not changed significantly from this, using larger buffers as increasing transistor budgets permit, able to track approximately 50 instructions. The logical complexity consumes significant transistor resources, so is generally not present in low-cost or low-power processor designs.

2.1.3 Multi-core processing

Dennard scaling was the primary driver of improvements in performance of MOSFET processors. This law states that as transistor size reduced, their power density was unchanged, so that voltage and current consumption each reduced proportionally to length [17]. This enabled chips to contain more transistors and operate at higher clock frequencies without increasing total chip power consumption, which is typically a hard limit. This law began to noticably break down as transistor lengths approached 65nm in 2005, as current leakage became a significant problem through wasting 50% of total power. Thus the useful power needed to be utilised more efficiently, and as power was proportional to the square of frequency then frequency increases had to end. New transistor designs and fabrication techniques were able to reduce current leakage, but the increased headroom was used for more transistors. Post-2005 processors continue to increase transistor counts, but have seen their operating frequencies stagnate; to continue improving performance they transitioned from single-core to multi-core designs.

Where a "single core" processor contains a single core (the instruction pipeline) and one or two levels of cache memory, a multi-core processor contains multiples of these within a single silicon package (an example of multiple instruction, multiple data (MIMD)). A cache coherence protocol such as MESI is implemented to allow multiple cores to safely read/write on the same data (cached in their respective local caches) [46], and a typical multi-core processor has an additional level of cache shared between the cores. Figure 2.3 illustrates a typical server-grade multi-core processor, with many cores communicating over an interconnect bus, an additional level of shared cache, and an additional IMC to increase memory bandwidth.

For a computational workload to utilise multiple cores to increase its throughput, it must instantiate multiple and parallel strands of execution then distribute its workload among them. The former can be achieved using lightweight threads, each

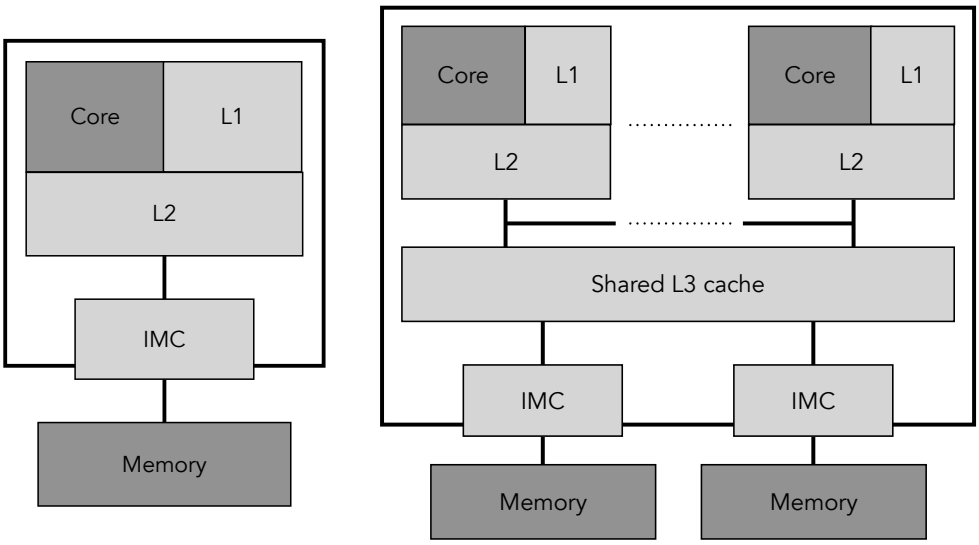


Figure 2.3: Comparison of single-core (left) and multi-core (right) design. Single-core designs have one IMC and two levels of cache. Multi-core typically has a third level of cache shared by all cores, and server-grade designs have multiple IMCs for more bandwidth.

executing on a separate core; threads share the same address space, which provides an opportunity for greater data reuse; but this also limits them to reside within the same operating system session, in practice meaning the same computing node. An alternative to threads are processes, each of which has a separate and independent address space; typically these are used when threading is not possible, such as in a distributed memory system. Workload distribution requires that the input data arrays be partitioned, and that each thread or process be allocated one partition. Additionally, if the computation kernel is stencil-based, then data communication must be performed between cores of adjacent partitions.

2.2 Performance profiling

Performance profiling is the targeted collection of fine-grained performance data during code execution. Typically this is measurement of time duration using system timers, known as walltime, but can also include hardware performance counters that count events such as CPU cache hits and misses. There are two common methods for collecting this data, with different tradeoffs – sampling profilers, or source code instrumentation. A sampling profiler is particularly useful when source code is not modifiable, or when a quick analysis is required. At frequent intervals the profiler

interrupts execution of the target code to record the executing function and its call stack, and may also record performance counter values. With sufficient sampling, the total runtime can be partitioned to the observed executing functions. Data accuracy is dependent on the frequency of sampling, and commerical profilers such as Cray CrayPat and Intel VTune use a default sampling interval of 10ms.

If fine accuracy or granularity is required but without the overhead issues of very frequent sampling, then the alternative to the sampling profiler is source code instrumentation. This is the insertion of data-collecting instrumentation directly into the source code of a target application, typically consisting of additional function calls to an external data aggregator that records provided data and finally writes to file. The direct insertion allows (i) the collected performance data to be directly associated with specific code blocks, (ii) for overhead costs to be controlled, and (iii) for non-performance data to be collected such as array sizes and variable values. Thus this provides the most accurate data, but for a comprehensive understanding then significant code modification is required; for a large application, comprehensive collection is only feasible with a tool that automatically inserts the instrumentation. Consideration must be given to the overhead of the instrumentation, and so is typically inserted around significant loops and pre-existing function calls rather than within.

Whichever method is used, the collected data is at the fine granularity of individual functions or loops, and for a parallel code the data is further broken down to individual processes or threads. This enables an understanding of which portions of the code are consuming the most time, and whether a parallel workload is being evenly distributed across processes. Further, it enables the calculation of performance metrics that concisely represent application performance, and enable comparison with other codes.

2.2.1 Runtime metrics

When analysing runtime, or using it to inform a decision, it is typically compared against a second runtime after a change in execution environment. This could be a change in hardware such as a processor upgrade, or a change in the software such as a data layout optimization or different parallel programming abstraction. The most common metric here is runtime speedup, which for example is 2× if the runtime reduced by 50%:

$$speedup = \frac{runtime_{before}}{runtime_{after}} \tag{2.1}$$

This can also be applied to a change in parallel hardware resources assigned to execution of a parallel code. Specifically, the speedup provided by N parallel threads over one serial thread can be defined as:

$$S(N) = \frac{runtime_1}{runtime_N} \tag{2.2}$$

This can be generalised to measure the speedup of N parallel *resources* over one; this resource can be one thread as describe above, or it can be one entire compute node. A better metric for parallel performance incorporates the expectation (or hope) that speedup will equal N. If the achieved speedup is less than N, then the additional hardware resources are not being fully utilised. This metric is parallel efficiency, and is calculated by dividing the speedup by N, giving a value between 0 and 1 (full efficiency):

$$efficiency(N) = \frac{S(N)}{N} \tag{2.3}$$

There are two common reasons why parallel efficiency can be below 1.0, otherwise known as parallel efficiency loss – resource contention and imperfect workload decomposition. Most multicore systems have two shared hardware resources - main memory, and the network interface. The total electricity consumption and heat dissipation within a processor package can also be considered a shared resource. This manifests as a problem in multicore processors with dynamic clock frequency - as more cores are utilised then heat and power draw increases, and the clock frequency is reduced to manage these. This is also known as thermal throttling, and is particularly noticable with vectorised codes. For many HPC codes and systems, contention for these shared resources is a significant problem and constrains parallel performance.

The second reason for parallel efficiency loss relates to how a parallel workload is distributed and synchronised among threads or processes. The goal with parallelising a workload is that each parallel thread of execution receives an equal portion of work, such that they complete at the same time - a perfectly balanced load. Any inequality in the distribution results in some threads idling as they wait for other threads to finish - load imbalance. Minimising load imbalance is particularly difficult for some codes, and an important area of research. Most HPC codes that distribute work among processes (rather than purely threads) will require frequent inter-process communication to synchronise data. For example with iterative stencil loops where an array value update requires neighbouring values, some of those neighbours will reside on other processes requiring inter-process communication to transfer their

values. This synchronisation introduces additional work that can become significant at high process counts.

2.2.2 Performance counters

Within processors there are hundreds of specific and varied hardware events. Some common examples are a clock cycle, the retiring of an instruction, and a L1 cache miss. There are more specific events, such as a clock cycle on which no instructions were scheduled for execution, or a memory store instruction being retired. These allow for a uniquely deep insight into performance of a code. To count occurrences of these events a modern processor provides a much smaller number of specialised registers. These counters are directly integrated into the architecture, so they count events with zero overhead; only the configuration and later retrieval from user space has overhead, albiet low at approximately 10,000 cycles, which for a benchmark code like STREAM is 5% or less [50].

Configuring of and access to most counters is performed via model-specific registers (MSRs), which are only accessible from kernel space; the exception are uncore (off-core) events that are configured via the PCI address space, but in Linux this is also only accessible by the kernel. The `perf` interface in Linux provides user access to this functionality, as both a command-line utility and through an API of system calls. On top of this are built several libraries, the most popular being the performance application programming interface (PAPI) library [7]. It provides access to 'preset' events, which are a commonly-used subset of all available CPU core events with architecture-agnostic and vendor-agnostic aliases; this set includes the example hardware events listed earlier. It also provides access to all 'native' events, necessary when measuring uncore events such as those monitoring main memory traffic. Listing 2.1 shows the ease with which PAPI allows several performance counters to be monitored during execution of a code block, in this case cycle consumption and number of instructions of a computation loop.

Listing 2.1: PAPI code to record cycles and instructions of 'compute_loop()'

```
int event_set_id;
PAPI_create_eventset(&event_set_id);
PAPI_add_named_event(event_set_id, "PAPI_TOT_CYC");
PAPI_add_named_event(event_set_id, "PAPI_TOT_INS");
PAPI_start(event_set_id);
compute_loop();
long counts[2];
PAPI_stop(event_set_id, counts);
```

There are many events within a modern processor that can be monitored, many of which have configurable options. A summary of the most useful events are covered here, many of which relate to the cache hierarchy. The L1 cache typically has a small number of events, for when a cache line is replaced. A line replacement indicates that a load instruction requested data not in cache, or that the hardware prefetcher pre-emptively loaded a new cache line. The remaining levels of cache have more events, with fine granularity to provide a rich understanding of how the cache hierarchy is being utilised. They differentiate between requests for code or data, whether a request is to load data or read-for-ownership (RFO) (for writing), whether a load/store instruction or prefetcher created the request, and whether the requested data was present or missing. The core pipeline has various events being monitored, in particular: cycle consumption, stalled cycles, and instruction counts.

2.3 Mini applications

There are numerous benchmarks and mini-applications representing the performance of different classes of HPC applications, some of which have been released as component parts of projects such as the Mantevo Project [25], the ECP Proxy Apps Suite [1], and the UK Mini-App Consortium [14]. Mini-applications from these repositories have been used in a variety of contexts.

One such example is miniMD, which has been used to explore the performance of molecular dynamics codes on the Intel Xeon Phi Knights Corner architecture [47]. Using a combination of AVX intrinsics and algorithmic optimisations, e.g. overlapping PCIe transfers with computation, the authors demonstrate a 5× speed-up for the gather-scatter bottleneck typically present in MD codes.

Mallinson et al. compare the performance of two PGAS programming models (OpenSHMEM and Co-Array Fortran) against MPI using CloverLeaf, an Lagrangian-Eulerian hydrodynamics mini-application [36]. The authors demonstrate that OpenSHMEM is able to outperform an equivalent MPI implementation by 7.78 iterations/sec, at 4096 sockets, when using proprietary nonblocking operations from Cray and 4 MB memory pages.

LULESH, a hydrodynamics mini-application representative of ALE3D, is used to assess the suitability of emerging parallel programming models (e.g. Liszt and Loci) along with more established models such as OpenMP [31], in terms of programmer productivity, runtime performance and ease of optimisation. The reduced size of LULESH when compared with ALE3D allowed the authors to examine eight parallel programming models. Their conclusion highlights that while the emerging models

such as Chapel and Loci enable a high level of productivity, they cannot match the performance of more established models such as MPI and OpenMP.

Similarly, Giles et al. examine the performance of OP2, a domain-specific abstraction library (DSL) for unstructured grid codes using the AIRFOIL CFD mini-application [48]. The authors demonstrate that they are able to achieve performance within 6% of a hand-coded implementation.

The CFD code included in the Rodinia benchmark suite has been used to examine the performance of a Graphic processing unit (GPU) when running unstructured grid applications [15]. From the results, Corrigan et al. conclude that GPUs show promise for this class of code given an increase in double precision performance in the future.

2.3.1 Characterising similarity

A typical mini-application is designed around one or several key loops extracted from a target application, i.e. it contains code from the application with minimal 'boilerplate' for execution (hence "mini"). Thus it can be assumed that the key performance characteristics are captured, with some simple performance comparison to verify this. But it is not always possible or desirable to share code with an application, for example if that application is subject to intellectual property (IP) restrictions that greatly limit portability – one important feature of a mini-application is portability. For these scenarios, the mini-application will have to differ in code, but still capture the key performance characteristics – this sub-class of mini-application is referred to as a proxy-application. Then verification of similarity becomes more important.

Research on verifying similarity is a relatively young field, with the earliest known work in 2014. Most of this research has a common theme – quantify the degree to which a proxy-application and its target application stress the same parts of hardware, and design a metric to embody the similarities. Barrett et al. proposed that if a proxy-application is similar to its target, then it should be predictive of performance [4]. A direct interpretation of predictive is used, meaning the performance metric (e.g. runtime) should simply match without transformation. Tramm et al. first proposed using correlations between hardware performance counters and parallel efficiency loss, where a strong correlation indicates a particular hardware event is harming performance [59]. Where strong correlations are exhibited (> 0.85), if both codes exhibit similar correlations for the same hardware events, then the proxy-application can be considered to be similar to the target. A limitation of this work is that the data processing is manual. Is 0.85 the right threshold for a

strong correlation? What size difference between the two correlations with an event is needed to mean dissimilarity?

Islam et al. improved upon this approach with machine learning techniques, automating feature selection and data analysis [29]. This enables two robust metrics: (i) the significance of a particular hardware resource (e.g. L2 cache) to the selected performance attribute (e.g. parallel efficiency loss); (ii) the similarity of this significance between proxy-application and target code. Aaziz et al. added MPI metrics to the analysis – times, message sizes and counts, grouped by MPI primitive type (send, recv, collective, wait) – and applied hierarchical clustering directly to the metrics rather than correlate with runtime or scaling [2]. A standard cluster similarity metric quantified similarity of a proxy-application, effectively for in-core metrics, but struggled on their MPI metrics where the proxy-application used different MPI primitives to its target. They later abstracted away from primitives with a metric based on pair-wise communication, assisted by CrayPat profiler [3].

2.4 Performance modelling

The previous section summarised research on verifying proxy-applications. These are important and useful contributions, adding robustness to the verification. But they do not directly address whether or how a proxy-application can predict performance of its target application. If a proxy-application is highly similar to its target application, then it should be possible to use it to generate accurate performance predictions of the target. Two relevant areas of research are reviewed here – performance projection, and mechanistic modelling of processors.

2.4.1 Performance projection

Performance projection concerns the application of purely statistical or machine learning techniques, to predict performance of one code from a variety of others. Sharkawi et al. propose a technique of identifying surrogate codes that are quantifiably similar to the target code according to 25 performance metrics [53]. These surrogates are executed, and a genetic algorithm calculates a weighting based on how similarly performance is affected by the hardware resources. Their weighted average is used as a prediction of compute performance (excluding MPI), achieving a mean error of 7.2% on a IBM Power6 and 10.5% on a Intel Core. In follow-up work they add a performance model for MPI communications [54]. Hoste et al. apply a similar technique but to microarchitecture-independent metrics, predicting the ranking of machine performance with 0.89 mean rank correlation [27]. Shweta et al. seek

to achieve both goals, predict performance (not just ranking) but without system-specific hardware metrics [52]. Similarity is calculated as in prior work, as Pearson correlation of a performance metric across systems, but they use collaborative filtering from social networks to identify similar benchmarks. Rather than predict runtime, they predict instructions-per-second (IPS). They report RMSE values, calculated across predictions – mostly these fall between 1 to 3, but the IPS values range 0.05 to 22.99, so for those applications near the low end of range this RMSE is very high. No per-application errors are given. Wang et al. apply a deep neural network (DNN) model to hardware features readily provided by manufacturers (cores, cache size, etc), to predict performance on new Intel architectures and SKUs [64]. Evaluated with two benchmark suites, SPEC and Geekbench, model error is 5% and 11% respectively.

2.4.2 Mechanistic modelling of superscalar processors

A mechanistic core model is an analytical performance model with structure derived from the processor core architecture. In the context of modern superscalar processors, this model typically focuses on throughput of instructions through, and bottlenecks of, the various pipeline stages. These are intended to execute several magnitudes faster than a cycle-accurate simulator.

Michaud et. al present the first mechanistic model relevant to a superscalar processor, constructing a simple model of the interaction between branch misprediction rate and instruction fetch rate, exploring how this affects the achievable ILP within the processor design space [38]. Despite its simplicity it provided useful insight into processor design, indicating that to double ILP it is necessary to both double the instruction fetch rate and reduce the branch misprediction rate by 4×. Taha et. al extend this much further to cover more stages of the superscalar pipeline [56]. They also introduce the concept of the *interval* model, although they term this *macroblocks*, where instructions are approximated as executing in blocks separated by misprediction-induced stalls. They implement three specific extensions to capture additional bottlenecks within a superscalar processor:

- limited throughput of functional units of each instruction class
- reorder buffer with limited size, which limits the scope for out-of-order execution
- limited instruction retirement bandwidth

The resulting analytical model does have a simulation aspect to estimate the ILP of a code, treating this as an emergent property of the interaction between code and architecture. This simulation tracks chunks of instructions flowing from the cache

through the queue, the reorder buffer, the functional units, then finally to retirement. This approach is both coarse and abstracted, avoiding the computational costs of cycle-accurate simulators. Transfer between each stage is bounded mechanistically; of particular interest is the formulation for the functional unit bottleneck. This states that the overall instruction issue rate (to functional units) is bounded by that class of instruction which has the lowest ratio between number of compatible functional units and proportion of executing code that is of that class. Validated against a cycle-accurate processor simulator of a MIPS R12000 out-of-order processor predicting the performance of the SPEC95 integer and floating-point benchmark suite, the model completed 40,000× faster and diverged with mean absolute error of 5.5%. Of particular interest are three of the SPEC95 floating-point benchmarks that solve PDEs, as these are similar to the code that is the subject of this thesis. For these benchmarks, the mean error is higher at positive 12.8% with a bias to over-prediction.

Van den Steen et al. apply a similar mechanistic model to the Intel Nehalem architecture [61]. Much of this work is to evaluate architecture-independent models of cache and branch miss rates on an existing interval model [19], but they also propose an improvement for functional unit modelling. One key difference in Nehalem from the older MIPS 12000 is that it attaches multiple functional units to a single port, in particular just two ports host 16 of the 20 of the arithmetic functional units [28]. In contrast, within the MIPS 12000 architecture each functional unit is located on a single dedicated port [65]. This introduces a new potential bottleneck, which they model with no abstraction with the rule that each port can accept at most one instruction per cycle, under the condition that the target functional unit is idle. Validation against the SPEC CPU 2006 benchmark suite produced similar accuracies as Taha et. al, with mean absolute error 7.6%, and for the four floating-point benchmarks that solve PDEs mean error is positive 12.5% with a bias to over-prediction.

2.5 Summary

This chapter provides an overview of the various degrees of parallelism within a typical modern computing cluster node. Achieving a high proportion of the available performance requires thought and consideration from the HPC programmer; it is insufficient to rely on the compiler to extract all parallelism from a code. Particularly, thought must be given to how data is mapped to the parallel hierarchy.

This chapter also covers tools and techniques that help navigate the complexity in HPC. These evaluate whether a HPC code is extracting a good proportion of the available parallelism. They also help to identify performance improvements, whether through code optimisation or targeted hardware upgrades. Finally, it summarises current research in performance prediction, which can improve the benchmarking capability of these tools.

Chapter 3

Computational Fluid Dynamics, Software and Hardware

CFD is the use of numerical analysis to solve a system of mathematical equations that describes physical phenomena relating to fluid dynamics. The development of CFD can be traced back to Los Alamos National Laboratory in the late 1950s, when they received a sufficiently capable scientific computer [24]. CFD developed with increasing computing capability, enabling the simulation of increasing complexity in fluid flow and geometry.

3.1 HYDRA

HYDRA [33] is a suite of nonlinear, linear and adjoint solvers developed by Rolls-Royce plc. in collaboration with a number of UK universities. These solvers target airflow within turbomachinery, where the flow must be modelled as compressible, viscous and turbulent. As such they solve the Reynolds-Averaged form of the compressible Navier-Stokes equations, which embody conservations of mass, momentum and energy. Turbulence modelling is enhanced with the Spalart-Allmaras one-equation model [55]. Equations are discretised using a MUSCL-based flux-differencing scheme, then block Jacobi preconditioned [39]. An explicit 5-stage Runge-Kutta scheme is applied to improve stability in high viscosity regions, and convergence of the multigrid method [37].

3.1.1 Unstructured grid

When seeking to model fluid flow, there is a physical geometry which it flows through or around, such as an airfoil or a turbine blade. A critical decision is

Table 3.1: Significant constituents of HYDRA runtime, measured on a single node of Xeon Broadwell with 28 MPI processes

Loop	**Function**	**Runtime %**
VFLUX	Viscous fluxes	35.8
GRAD	Gradient	16.8
SRCSA	Spalart-Allmaras source term	14.6
IFLUX	Inviscid fluxes	10.7
UPDATE	Update flow	7.3
JACOB	Jacobi preconditioner matrices	6.8
—	Other routines	8.0

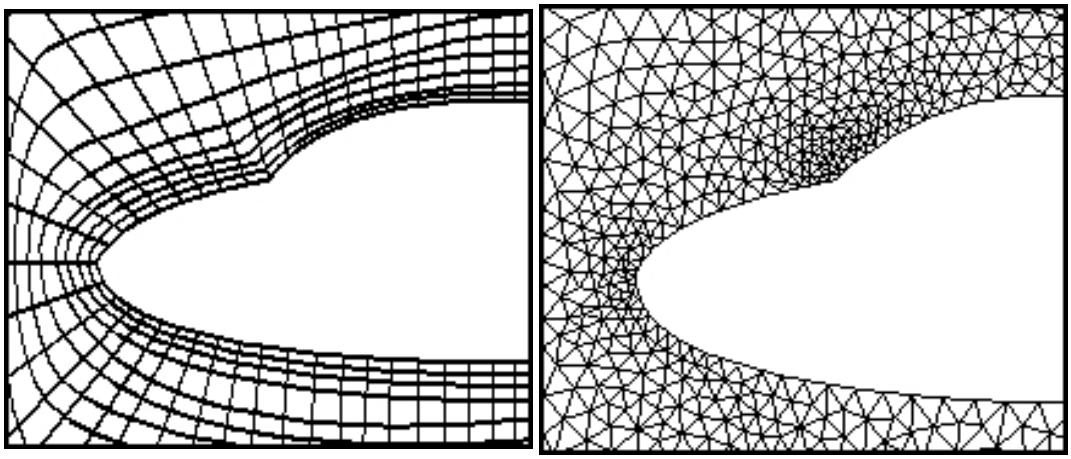


Figure 3.1: Comparison of structured and unstructured grid [62].

whether to represent this geometry and surrounding space using a structured grid or an unstructured grid, as it influences numerical accuracy and computational performance.

A structured grid is generated by decomposing a spatial volume into cells according to a fixed and regular topology that maps directly to array elements. Then the spatial properties of an array element (position, volume etc) and its neighbours can be determined solely from the topology, and do not need to be stored in memory. Assuming a 3D space, then the simplest topology is a 3D arrangement of identical cuboids that map directly to a 3D array. A structured grid has limited ability to represent complex surfaces, if those surfaces can be accurately represented with parametric curves or surfaces. For example in Figure 3.1 (left), a structured grid is representing an aircraft nose and cockpit with parametric curves.

An unstructured grid uses no topology to decompose space. Instead, it partitions space into polyhedra cells with no constraints on size, uniformity, nor regularity. A mesh of nodes and edges is fitted to the cells, such that each mesh node represent a cell and each mesh edge represents the shared face between a pair of adjacent cells. With no topology restricting decomposition, an unstructured grid is better able to represent complex geometries and adaptively increase grid resolution in spatial regions where fluid flow is potentially more complex or of more importance. An example is shown in Figure 3.1 (right). The cost of this freedom is greater difficulty in achieving high-performance, particularly around memory utilisation. Consider a typical CFD calculation, calculating and accumulating fluxes passing into a cell through its faces – this can execute as a unit-stride loop over mesh nodes (cells), that then must *indirectly* access data belonging to connected mesh edges (cell faces) such as area and the flux. An indirect memory access occurs where the memory address cannot be directly calculated from the loop iterator, but instead is an arbitrary value stored in memory, for example `v = e[n[i]]`. This indirection leads to irregular memory access, that reduces spatial locality and the effectiveness of hardware prefetchers. A third approach is to use a hybrid grid, best thought of as an unstructured grid of structured grids. This Thesis concerns a CFD application that uses unstructured grid, and so further discussion or reference to grid representation will focus on the purely unstructured variant.

In this work two different meshes are used of very different sizes, both multigrid (MG). The smaller mesh is derived from the geometry of Whittle Laboratory's low pressure axial flow turbine rotor cascade, a mesh of 105 K nodes and 305 K edges representing a single rotor root section (blade and hub connection) [26]. To aid visualisation a rotor section of NASA's SSME 2-stage fuel turbine is shown in Figure 3.2, consisting of multiple root sections with similar structure to the mesh used [42]. Three MG meshes are derived from the base mesh, introducing an additional 118 K nodes and 439 K edges. Being a small mesh, this is particularly useful for weak scaling where a larger mesh would exceed available memory.

The larger mesh used is the NASA `Rotor 37` mesh of an axial compressor rotor [49]. The geometry it represents is also similar to the example rotor shown in Figure 3.2. This contains approximately 8.1 M nodes and 24 M edges, with an additional three MG meshes that results in a total count of approximately 15.7 M nodes and 53 M edges. This mesh is best used for strong scaling benchmarking, of up to several hundred processes.

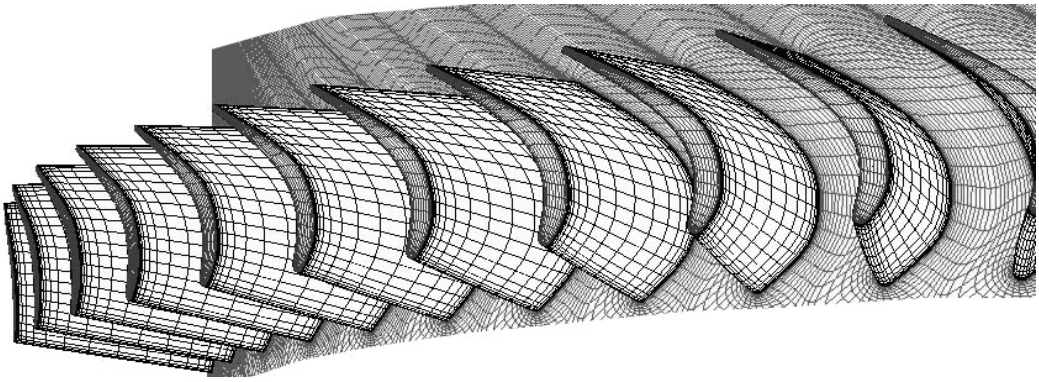


Figure 3.2: Visualisation of a rotor section from NASA's SSME 2-stage fuel turbine, similar to the meshes used in this work.

3.1.2 Multigrid

HYDRA employs multigrid methods which are designed to increase the rate of convergence for iterative solvers, and possess a useful computational property – the amount of computational work required is linear in the number of unknowns [60]. Multigrid applications operate on a hierarchy of grid levels; in this paper, we are concerned with geometric multigrid, wherein each grid level has its own explicit mesh geometry, and the coarse levels of the hierarchy are derived from the geometry of the finest level. One method of constructing a coarse level is to join pairs of cell volumes in the finer level, as shown in Figure 3.3. This chosen construction method determines how information is transferred between cells of adjacent levels, with each cell of the coarser level being linked to those cells in the finer level from which it was derived.

Starting at the finest level, multigrid applications use an iterative *smoothing* subroutine to reduce high frequency errors. Low frequency errors are then transferred to the next coarsest level (*restriction*), where they appear as high frequency errors and can thus be more rapidly smoothed by the same subroutine. Error corrections from the smoothing of coarse levels are then transferred back to finer levels (*prolongation*). The order in which prolongations and restrictions are applied is know as a cycle, of which this paper considers a single type – the so-called V-cycle.

There are several performance implications of using a geometric multigrid solver. First, there is the increased memory requirement of explicitly representing the geometries of all levels of the multigrid. Second, there are the additional irregular memory accesses from prolonging and restricting corrections between levels of the multigrid. Third, the coarsened meshes have reduced spatial locality.

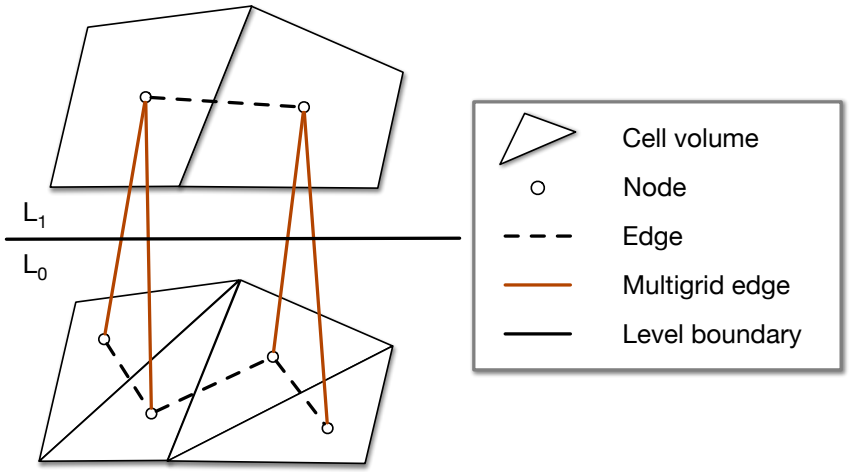


Figure 3.3: Representation of a finite-volume decomposition mapped to an unstructured grid over two multigrid levels.

3.1.3 OPlus

Originally a sequential code, HYDRA incorporated the OPlus DSL to provide parallel computation [11]. OPlus (Oxford Parallel Library for Unstructured Solvers) is designed to be inserted into an existing codebase with minimal effort, if of course that code is concerning computation over unstructured grids. The programmer adds calls to OPlus routines around each compute loop, with the library itself managing all parallel-related activities, namely the partitioning of the input grid, file I/O, and inter-process data synchronisation. This enabled HYDRA to transition to the highly parallel machines that began to emerge several years prior.

3.2 HYDRA performance engineering

3.2.1 HYDRA performance model

An analytical performance model of HYDRA has previously been developed, most recently described in this publication [10]. It considers total runtime as being the summation of the maximum runtime of each loop, adopting the bulk synchronous parallel (BSP) model for loop execution across the MPI processes. Communication is assumed to overlap with the independent portion of compute.

This model adopts a bottom-up approach in calculating expected runtime, beginning with the average runtime of a single loop iteration known as *grind time*. As HYDRA operates on multigrid meshes, typically consisting of four levels of distinct

meshes with increasingly irregular sparsity, then grind time of a loop operating on different levels can (and often does) vary. Accordingly, each loop has four grind times, one for each multigrid level. In the model formulation this is termed $W_{g,l}$ (i.e. $W_{grind,level}$). Each process has N_i independent elements and N_d elements dependent on neighbouring processes data, determined by the mesh partitioning. Then the time spent in e.g. independent compute is simply $W_{g,l}N_i$. A simple network latency and bandwidth model is used to calculate the halo exchange time C for elements of size B bytes, with variables primed by the Intel ping-pong MPI benchmark: latency α and reciprocal bandwidth β. Then loop runtime W_l is the greatest of communication time C or independent compute, plus dependent compute:

$$W_l = \max(N_{i,l}W_{g,l}\ ,\ C_l) + N_{d,l}W_{g,l} \tag{3.1}$$

$$C = \alpha + \beta N_d B \tag{3.2}$$

The initial model predictions disagreed with observations, leading to the identification that the communication was not fully overlapping with the compute. Despite all MPI send and receive requests being initialised before computation, some MPI messages were not transferring until after the sending processes had completed their compute. The asynchronous `MPI_Isend` was replaced with its synchronous variant `MPI_Issend`, and an additional `MPI_wait` added, enforcing the behaviour that a process must wait until all of its sends have begun transferring before computing on its received dependent data. This modification provided a significant speedup, and the resulting observed performance was in agreement with the model prediction, shown in Figure 3.4.

Model limitations

The performance model has two limitations: the need to collect partitioning data, and the need to measure W_g on a target system.

For parallel execution N_i and N_d are determined by the partitioner, and although mesh partitioners aim to generate equal sized partitions, unpredictable variance in N_d across the processes is unavoidable due to the unique connectivity and geometry of the mesh. This variance is an increasing problem for predicting strong scaling performance, as the size of the variance relative to N_i increases. Furthermore, N_i of each multigrid mesh is determined by its relation to the partitioned finest mesh, not directly by the partitioner, which is observed to lead to significantly unequal partition sizes. Thus to generate model predictions, the mesh must be actively

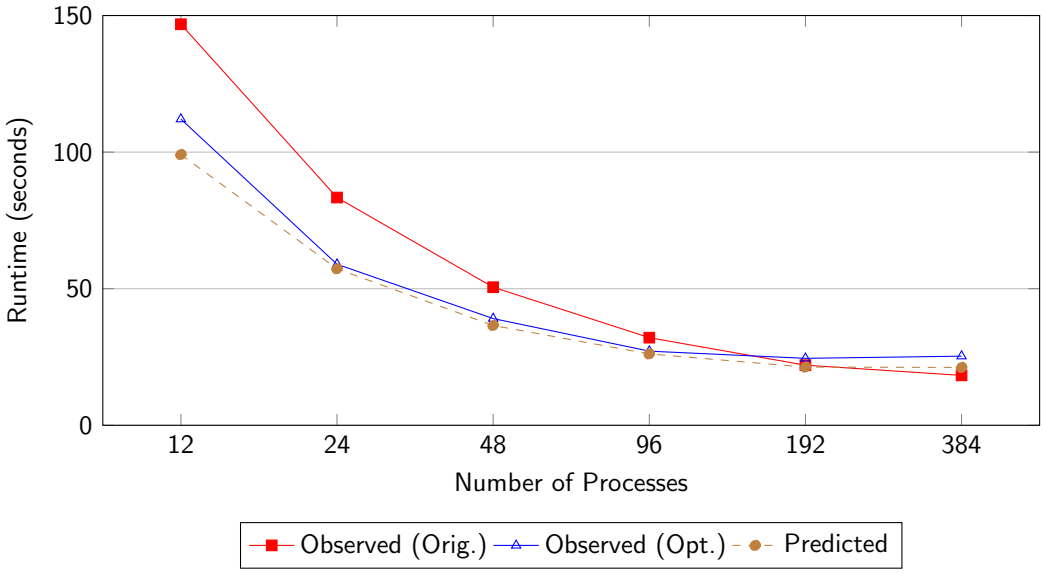


Figure 3.4: HYDRA observed performance, before and after optimisation, and the performance model predictions. Model highlighted sub-optimal performance, caused by nonblocking asynchronous MPI communication not overlapping with compute. Switching to nonblocking *synchronous* MPI improved overlapping, matching model.

partitioned at the desired process counts, requiring sufficient hardware resources which may not be readily available.

Another limitation is that W_g must be measured, limiting this model to predicting performance on those systems that have been cleared for receiving and executing HYDRA.

3.2.2 MG-CFD proxy-application

A mini-application is a tool designed to capture one or several key performance characteristics of a target application, or of a class of applications. MG-CFD is one such tool, designed to capture the compute and scaling characteristics of the unstructured grid computation in HYDRA [8]. In the original work this software was named mini-HYDRA, only recently renamed to MG-CFD to emphasise that it does not contain any source code of HYDRA. It is based on an existing open-source CFD solver, rather than contain key CFD kernels from HYDRA, to avoid commercial portability restrictions [15]. Thus this makes MG-CFD a *proxy*-application, a subclass of mini-applications. This existing solver, now included in the Rodinia benchmark suite [15] [13], implements a three-dimensional finite-volume discretisation of the Euler equations for inviscid, compressible flow over an unstructured grid. HYDRA differs to this solver in only two key areas: (1) Navier-Stokes equations for viscous

and turbulent flow are discretised and (2) multigrid techniques are used to accelerate solution convergence. This existing code is written in C, and during adoption it was extended with simple use of C++ Standard Library (such as `vector` and `string`). This provides advantages over Fortran, as many contemporary HPC software technologies support C/C++ before Fortran. A notable example is NVIDIA CUDA, first released in 2007 with a C API with Fortran support following two years later [20]. The addition of C++ has been limited to Standard Library to ensure it retains the same degree of compiler portability as plain C.

Details

To capture the performance characteristics of multigrid in HYDRA, MG-CFD extends the open-source solver with crude multigrid operators to transfer grid state between levels. These operators are defined by Equations 3.3 and 3.4 which serve as restriction (fine to coarse grid) and prolongation (coarse to fine grid) operators respectively [6]. Where u_j^l represents simulation property u of node j at level l, and N_j^l is the set of node indices which are linked to node j at level l from $l-1$ of the grid.

$$u_j^l = \frac{\sum_{i \in N_j^l} u_i^{l-1}}{|N_j^l|} \tag{3.3}$$

$$u_{i \in N_j^l}^{l-1} = u_j^l \tag{3.4}$$

The restriction operator (Equation 3.3) primes the simulation properties with an average across nodes from the finer grid level – this mapping between levels is defined as part of the input deck. The prolongation operator (Equation 3.4) reverses restriction by injecting the values from the coarse grid to the fine grid as dictated by the mapping.

Validation

To enable validation of MG-CFD a mini-application was created, named compact-HYDRA, which contains an actual flux calculation kernel from HYDRA that has been ported from Fortran to C. Ideally the chosen kernel would be the viscous flux calculation kernel `vflux`, as it typically accounts for approximately 35% of total HYDRA runtime and is its single most expensive loop. However to simplify the extraction and conversion task, the smaller inviscid flux kernel `iflux` is selected. These two flux routines are computationally very similar, performing a calculation and integration of cell volume surface fluxes, but the `iflux` kernel performs much less

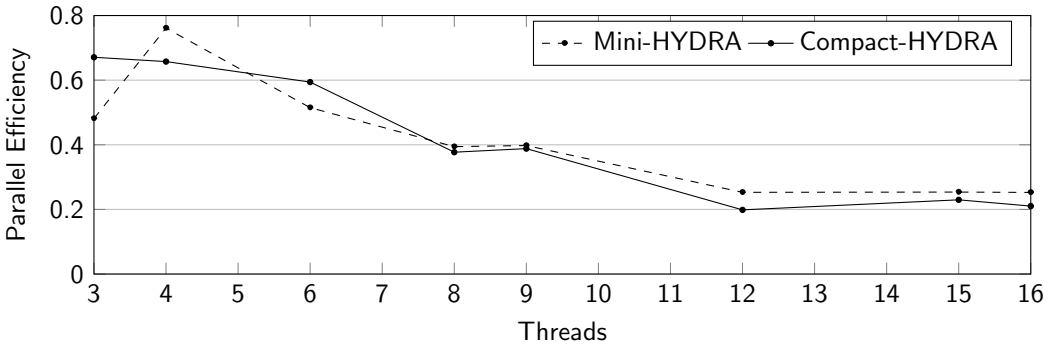


Figure 3.5: Parallel efficiency of mini-HYDRA and compact-HYDRA on Xeon Haswell.

arithmetic and so easier to detach from HYDRA. MG-CFD and compact-HYDRA differ only in the particular flux calculations performed – both loop over the same unstructured grid mesh, use the same multigrid cycle, and use the same Runge-Kutta time-stepping. This enables the attribution of any observed performance differences (or lack thereof) to the flux kernels.

MG-CFD is validated by comparing its scaling performance to that of compact-HYDRA. Firstly, parallel efficiency of each of the four multigrid levels is compared, with that of the finest grid shown in Figure 3.5. At low thread counts there is significant dissimilarity, with parallel efficiency of MG-CFD fluctuating with no trend while that of compact-HYDRA falls steadily with increasing thread count. The reason proposed was that MG-CFD is particularly sensitive to the available cache bandwidth, evidenced by it performing 2-3× more L2 cache reads than compact-HYDRA and that this fluctuation was not seen on the older Xeon Ivy Bridge architecture with 50% less cache bandwidth. At moderate and high thread counts both codes exhibit similar parallel efficiency, falling to a minimum of approximately 0.25. The other three grid levels exhibit a similar pattern.

The second validation method applied was to assess whether both codes share the same cause of parallel efficiency loss. This calculates the correlation between parallel efficiency loss and various hardware performance counters for each code, then they are manually compared and assessed [59]. Figure 3.6 presents the correlations on Xeon Haswell; for most counters, both codes exhibit a similar correlation, but there are two classes of counters where the codes are dissimilar. The first are L2 request counters (and L1 miss counters), where compact-HYDRA exhibits strong positive correlation but MG-CFD exhibits weak positive or strong negative correlation; this is attributed to both codes having different sensitivities to cache bandwidth. The second class relate to cache coherence events, with compact-HYDRA exhibiting no

correlation but MG-CFD exhibiting moderate negative correlation. This was not discussed, likely because the absolute values are below 0.8 and so not strongly related to parallel efficiency loss.

3.3 Summary

This chapter introduced HYDRA, a CFD simulation code used by Rolls-Royce for aerospace design engineering. CFD simulation is an important part of designing new and improved turbomachinery, in terms of power and fuel efficiency, so investigating HYDRA is beneficial. Its key performance characteristics are described – irregular unstructured grid, and more irregular multigrid, that map poorly to modern processor architectures, as these best process structured grid (simple arrays). An overview of how HYDRA currently achives HPC is provided, via the OPlus DSL.

Existing work on developing performance tools for HYDRA are summarised. An existing performance model of HYDRA can identify sub-optimal performance, and explore alternative partitioners, but it requires accurate benchmarking data. A proxy-application of HYDRA has also been developed and validated. Addressing limitations of the proxy-application, and exploring its use in collecting the benchmarking data, is the focus of this thesis.

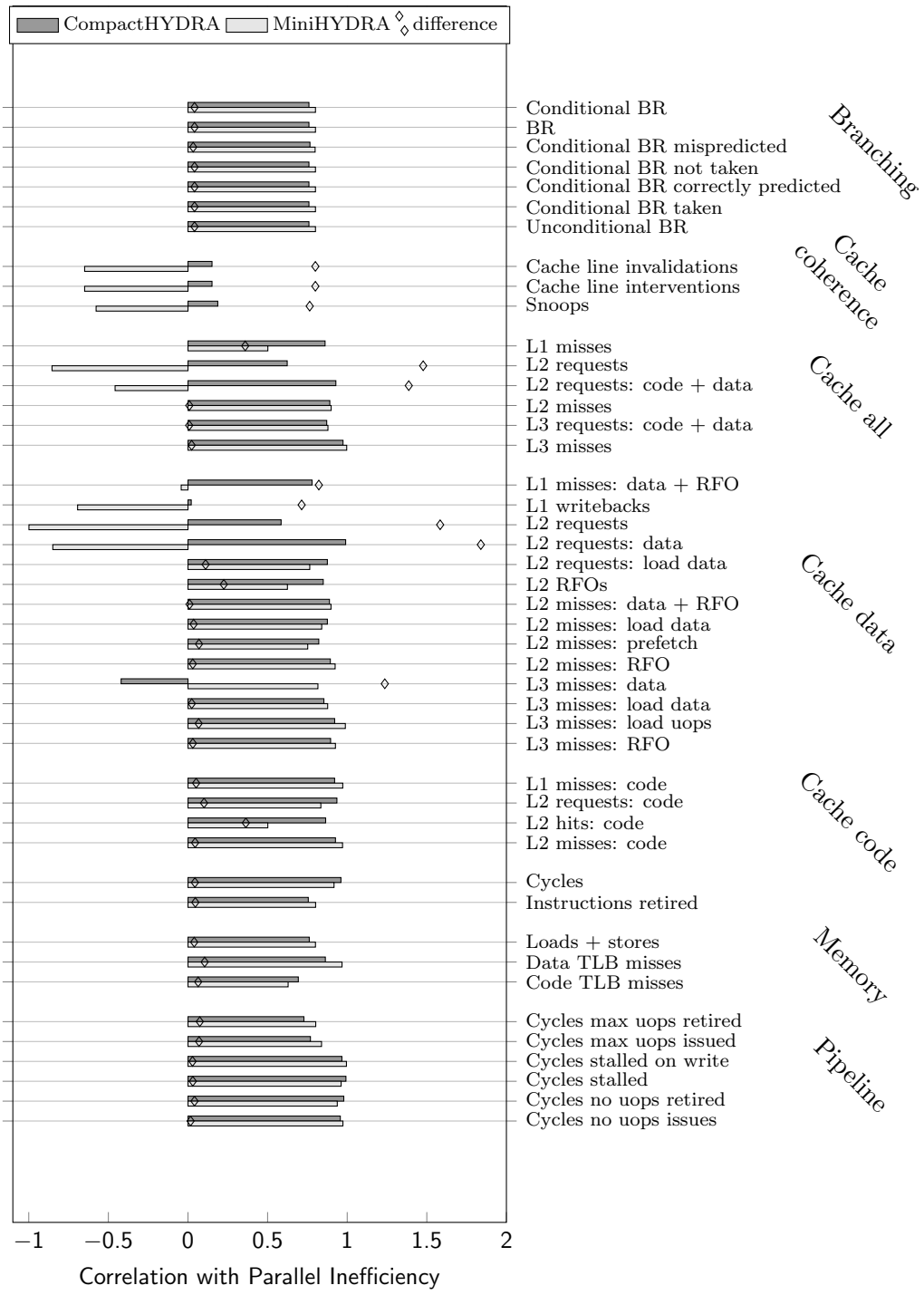


Figure 3.6: Plot of correlation between parallel efficiency loss and various PAPI performance counters for MG-CFD and compact-HYDRA on Xeon Haswell. The difference in correlation between MG-CFD and compact-HYDRA is also plotted, which can exceed 1.0 for serious divergence (worst case is ±2). Major divergence occurs with events relating to L1-L2 data traffic. Negligible divergence (i.e. high similarity) in events relating to L2-L3 traffic and pipeline utilisation.

Table 3.2: Hardware/software configurations

Hardware					
Architecture	Intel Westmere	Intel Broadwell	Intel Skylake	Intel Cascade Lake	Intel Knights Landing
Model	Xeon X5650	Xeon E5-2660 v4	Xeon Silver 4116	Xeon Gold 6252	Xeon Phi 7210
All-core GHz	2.8	2.1	2.7	3.2	1.3
Cores	12×2	14×2	12×2	24×2	64
Host ISA	SSE4.2	AVX2	AVX-512	AVX-512	AVX-512
Memory (GB)	24	128	96	384	16 HBM + 96 DDR
Software					
Operating System	Debian 8, Linux 4.9.0				
Compiler	Intel 18.0.2				
Tools	PAPI 5.5.1				

Chapter 4

Assessing and improving the proxy-application MG-CFD

Designing a proxy-application to be representative of a large simulation code is a challenging task. Without being able to simply extract the key loops from the target, but instead have a different codebase, it becomes critical to verify that the key performance characteristics have been captured. It is also important to consider whether differences between the proxy-application and target can create divergent results when implementing a code optimisation or hardware change.

This chapter reviews an existing proxy-application that has been validated using methods taken from literature. A careful investigation identifies deficiencies in the proxy-application that actually reduce its representativeness of the target application. These are resolved, and the proxy-application further enhanced to capture multi-node scaling characteristics.

4.1 Reviewing representativeness of MG-CFD

4.1.1 Arithmetic intensity

In the background Chapter 3.2.2 I presented the previous validation of the original MG-CFD proxy-application, then named mini-HYDRA. This exhibited similar parallel efficiency to compact-HYDRA, a Fortran-to-C port of a HYDRA loop selected for its relative ease of extraction, indicating that MG-CFD had captured the key scaling characteristics of unstructured grid compute. A necessary property of MG-CFD for it to exhibit this high similarity, particularly at high thread counts, is to have a similar arithmetic intensity as compact-HYDRA and so exhibit a similar dependency on memory performance. The code from which MG-CFD is derived

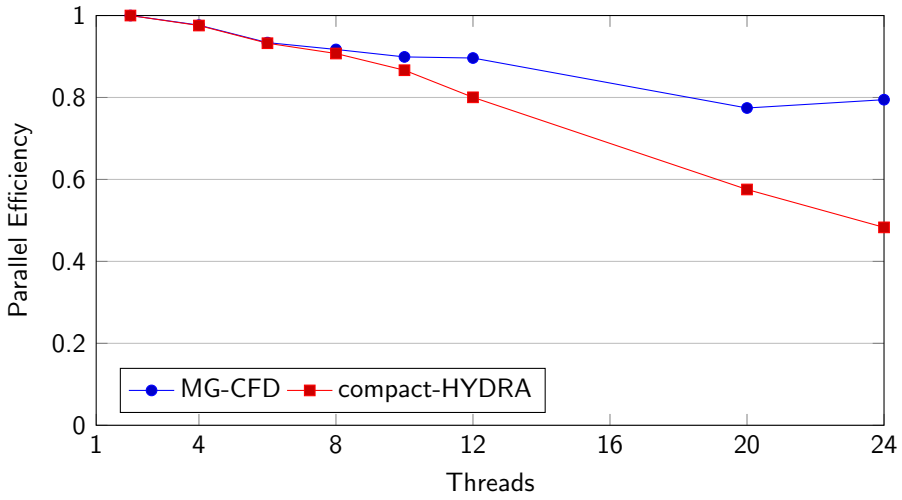


Figure 4.1: Parallel efficiency of compact-HYDRA and numerically-corrected MG-CFD, on Xeon Skylake with AVX-512 auto-vectorisation. Similar scaling exhibited until compact-HYDRA hits its memory-bound near 12 threads.

actually performs approximately 2× more arithmetic than compact-HYDRA, so an unmodified MG-CFD would not scale similarly. This divergence can be seen in Figure 4.1, which plots scaling performance on a Xeon Skylake node; initially scaling is very similar, but diverges once compact-HYDRA becomes memory bound. To equalise arithmetic intensity, the flux routine of MG-CFD had approximately half of its arithmetic removed artificially, breaking the CFD simulation aspect of the code. This then prevents its use for evaluating numerical optimisations such as mixed-precision floating-point, and raises the question of why base MG-CFD on a CFD code at all.

One solution is for MG-CFD to contain several variants of the flux kernel, each with a different arithmetic intensity to match a particular HYDRA kernel of interest; the generation of variants could be automated, so this could be a sensible approach. But for this approach to work it requires the assumption that the ratio between the instructions-per-cycle (IPC) rates of MG-CFD and the target kernel is invariant to system changes, meaning that the individual rates are invariant or that both codes are equally slowed-down (or sped-up). This is a reasonable assumption to make, as both kernels operate on the same unstructured grid, use the finite-volume method, and solve equations for inviscid fluid flow. However this assumption does not hold.

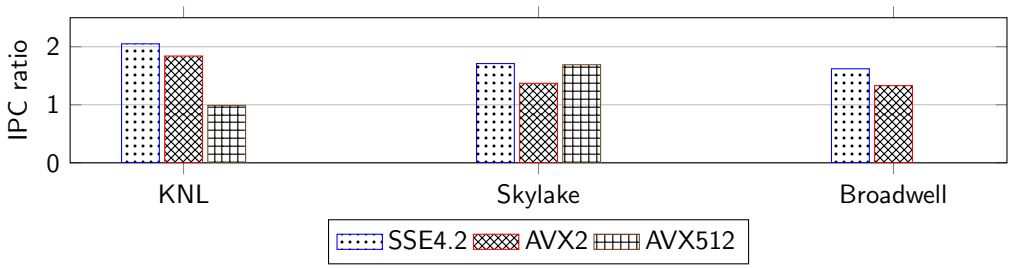


Figure 4.2: Ratio of compact-HYDRA IPC / MG-CFD IPC, of single-threaded execution across multiple ISAs and architectures. For MG-CFD to provide reliable performance assessment this ratio should be invariant, but variance exists - intra-ISA (AVX512 on KNL vs Skylake) and intra-architecture (AVX2 vs AVX512 on KNL (and Skylake)

Figure 4.2 shows the ratio between compact-HYDRA IPC and MG-CFD IPC across three architectures - Intel Xeon KNL, Skylake and Broadwell - and several ISAs. Varying both ISA and architecture increases the number of distinct execution environments in which to compare MG-CFD and compact-HYDRA. Execution is single-threaded to ensure it is assessing the architecture and not memory performance. The greatest ratio variance is observed within KNL, when switching ISA from SSE4.2 to AVX-512 (unvectorised). MG-CFD experienced a doubling of its IPC rate but compact-HYDRA experienced no change; an error of this magnitude when comparing systems is highly significant. Variance is also seen when changing architecture, from KNL to Skylake. Under AVX512, MG-CFD IPC increases by a modest 1.39× but that of compact-HYDRA increases by 2.38×, a significant speedup that would be missed if using only MG-CFD. A smaller but still significant disparity exists under AVX2, with compact-HYDRA IPC increasing by 2.75× but MG-CFD increasing by 3.69×, which would lead to MG-CFD overstating the benefit of a move from KNL to either Skylake or Broadwell. No significant changes are observed when switching from Broadwell to Skylake, which represents an incremental change in architecture. Skylake is the successor to Broadwell, so is a similar architecture with small tweaks regarding single-threaded execution, such as additional L2 cache and slightly improved out-of-order execution. KNL however is a very different architecture, based on Intel's low-power Silvermont architecture extended with out-of-order execution. Thus this indicates that while MG-CFD alone can accurately assess incremental architecture changes, it can not be reliably used to assess very different architectures, such as the Cavium ThunderX2 or AMD EPYC.

An alternative solution is to develop a performance model to describe the variance of IPC and account for differing arithmetic intensity, in terms of the few

static differences that exist between MG-CFD and compact-HYDRA. This can also provide deeper insight into performance than benchmarking, by highlighting those hardware features that are strongly bottlenecking performance. This can be useful in assessing the relevance of new features proposed in processor roadmaps, helping to pinpoint those future architectures worth obtaining evaluation access to.

4.1.2 Data-safe parallel computation

Another caveat with MG-CFD as originally designed is that its validation used strong scaling data, but ordinarily the indirect writes of unstructured grid codes would prevent data-safe parallel computation without a mesh partitioner. To address this, the original authors modified the flux routines of both MG-CFD and compact-HYDRA to write directly to a new intermediate array, with the indirect writes moved to a new subsequent sequential routine. This serves to reduce the similarity between MG-CFD and HYDRA, as HYDRA performs both flux calculations and accumulations together in the same loop. Specifically, this alters the interaction between the cores and the memory hierarchy, as direct writes map well to the memory structure which read/write entire cache lines, whereas indirect writes waste much of the available write bandwidth with written cache lines being mostly unchanged data. In the interest of ensuring that scaling performance data collected by MG-CFD is representative of HYDRA strong scaling, restoring these indirect writes is essential. To obtain data-safe scaling data without a partitioner, MG-CFD then must be weakly scaled, meaning that each thread of execution has one (or several) whole copies of the input mesh. At low thread counts, each thread may be given two or three copies of the mesh to ensure that cache locality effects do not disproportionally benefit their performances over many threads.

4.1.3 Validation of restored MG-CFD

To ensure that after these restorations that MG-CFD continues to be representative of compact-HYDRA, and by extension HYDRA, the validation process is repeated. As weak scaling is important here, then the mesh used is the smaller of the two meshes described in Chapter 3. This contains 105 K nodes and 305 K edges in the base mesh, and an additional 118 K nodes and 439 K edges in the MG meshes.

The PAPI library is used to collect performance counter data, which provides easy access to available performance counters and additionally defines a set of 108 “preset” counters that include performance counters typically found in many processors [7]. Figure 4.3 shows the correlation between each PAPI preset performance

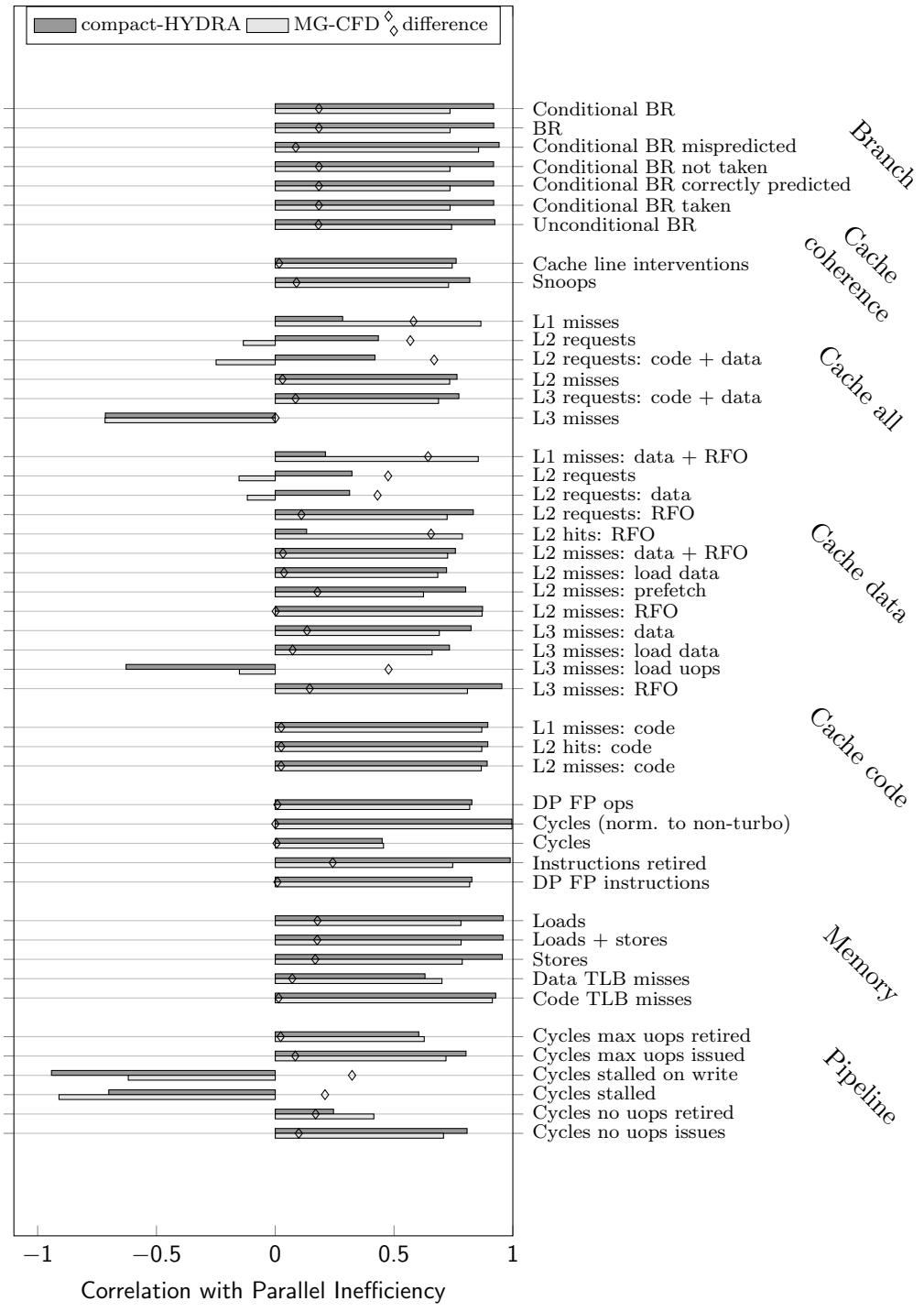


Figure 4.3: Plot of correlation between parallel efficiency loss and various PAPI performance counters for *restored* MG-CFD and compact-HYDRA on Xeon Skylake.

counter and parallel inefficiency. To account for variance of performance counters between runs the mean of three measurements is used. For most of these events the difference in correlation between MG-CFD and compact-HYDRA is less than 0.1, indicating that both codes share many performance characteristics, but there are several differences in correlations which we address here. The correlations for the memory events `loads` and `stores` differ by 0.2, with the correlation being stronger for compact-HYDRA. These events count store and load micro-ops, so compact-HYDRA being more sensitive to these is in agreement with it having the lower arithmetic intensity. A similar difference between correlations can be seen in the branching events, but neither code performs branching operations within the loop body so these are considered to be false positives. The only large difference is with events relating to L1 cache misses (`L1 misses`, and `L2 hits: RFO`), for which a strong correlation is only present with MG-CFD. This makes sense with knowledge that MG-CFD has register spilling, but not compact-HYDRA. With spilled registers effectively reserving some of the L1 cache, that leaves less for reuse of mesh data. The additional correlation with `L2 hits` rather than with all requests to L2 indicates that additional mesh data reuse is occurring from L2 not L1.

Where the correlation between a performance counter and parallel efficiency loss is greater than 0.8, this indicates that the corresponding hardware activity that triggers the counter has a strong influence on scaling performance. A strong correlation is seen with RFO requests and misses throughout the cache hierarchy. In the context of unstructured compute, this is an indication that the memory hierarchy is less able to adequately prefetch the destination arrays in advance of the indirect writes at higher thread counts. This in turn is an indication of contention in the memory hierarchy that is present with both codes. Other notable events are `Loads` and `Stores` which count store and load micro-operations and so is another indication of pressure on the memory hierachy.

4.2 MPI strong scaling

To achieve true strong scaling, with multiple threads of execution operating on parts of the same mesh, then mesh partitioning and exchange is required. Were the input a structured grid this would be simple, as the data arrays could simply be sliced. But with an unstructured grid, and particularly more so when that is multigrid, greater care is required to ensure that all of the indirect references are correct after partitioning and distribution.

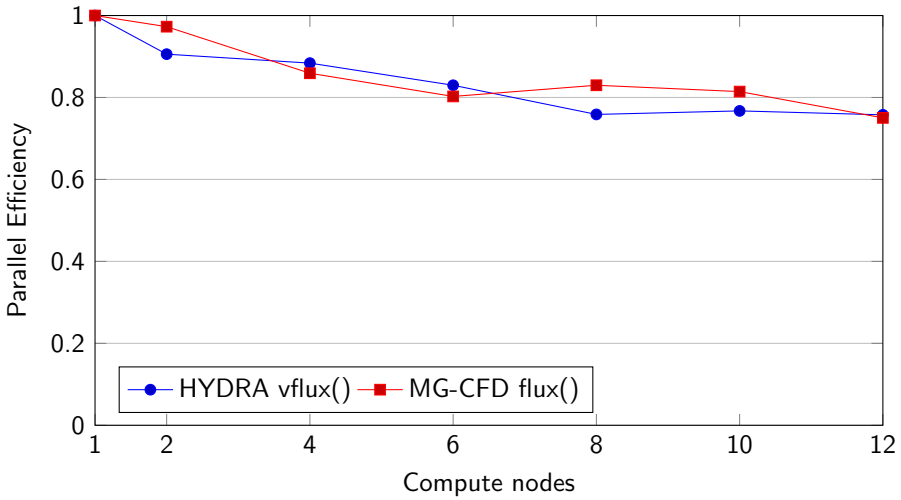


Figure 4.4: MPI strong scaling parallel efficiency on Westmere cluster, of the expensive HYDRA routine `vflux` and OP2-MG-CFD flux() routine

To reduce development time, an existing abstraction library designed for unstructured grid is used – OP2 DSL [41]. The key code integration point between OP2 and MG-CFD is the declaration of mappings between mesh elements – between edges and nodes, and between multigrid levels. Also for each loop, the data accessed and whether indirection mappings are used are declared to OP2. Then OP2 partitions the mesh using the selected partitioner (several are supported), distributes across parallel processes (or threads), and maintains data synchronisation during execution. OP2 DSL also includes a Python code generator for targeting other parallel programming APIs, such as CUDA and SYCL. Further justifying the selection of OP2 is that its developers also had significant involvement in developing the OPlus library used in HYDRA, which also partitions and distributes the mesh. The continued similarity in code should result in continued similarity in scaling performance of MG-CFD and HYDRA.

This version of MG-CFD with OP2 integrated will be referred to as OP2-MG-CFD. With strong scaling in place, then validation can take place across a cluster of nodes rather than a single node. The particular cluster used here is consists of 12 nodes connected by Infiniband, and each node contains 12 Intel Westmere cores – full node details are detailed in Table 3.2. With more cores, it is appropriate to use a larger mesh than was used in the previous validation – the `Rotor 37` MG mesh with approximately 24 M nodes and 77 M edges across four MG levels.

This validation is performed directly against HYDRA, rather than compact-HYDRA. Parallel efficiency is also calculated slightly differently; rather than use one

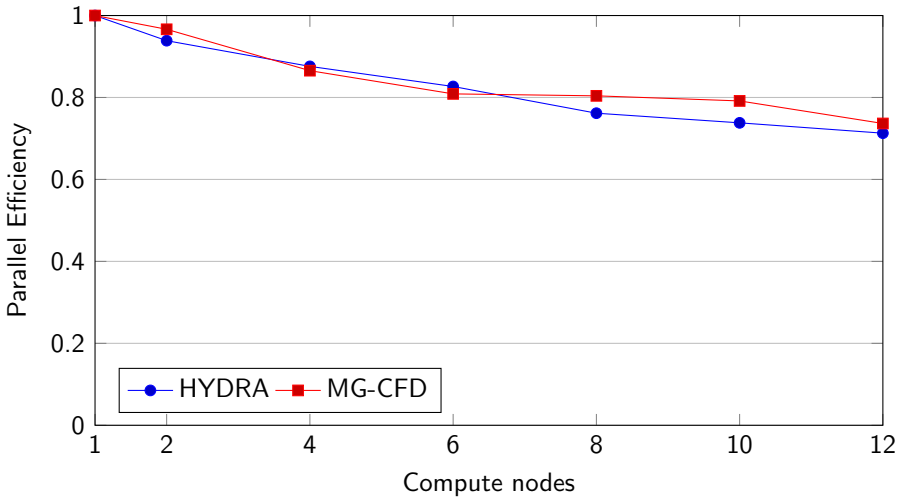


Figure 4.5: MPI strong scaling parallel efficiency on Westmere cluster, of total walltime of HYDRA and OP2-MG-CFD

thread/process for the baseline performance, instead one fully-populated cluster node is used, with multi-node scaling being compared. First comparing specific kernels, Figure 4.4 shows the parallel efficiencies of (i) the flux routine in OP2-MG-CFD, and (ii) the `vflux` routine in HYDRA, its most expensive loop. The `iflux` routine in HYDRA is also shown, which was previously ported into compact-HYDRA. Across all cluster node counts, OP2-MG-CFD flux() and `vflux` achieve a very similar parallel efficiency, and at the highest count the difference is small enough to fall within variance. The same high similarity is seen when comparing parallel efficiency of total walltime of each code, shown in Figure 4.5. This high degree of similarity is expected, as both codes were operating on the same mesh geometry, which through the partitioner determines the load imbalance and communication pattern.

4.3 Summary

This chapter identifies the issues that can arise when using a proxy-application to make inferences of the expected performance of a target application. Relatively minor differences in the code content can result in significant differences in performance when assessing new architectures. With the use of mini-HYDRA to directly infer performance of compact-HYDRA, this issue manifested in two ways: (a) different overall IPC, and (b) different memory-bounds. These manifestations would greatly impair the ability to predict speedup of a new architecture or computing node. In the original development of MG-CFD, then named mini-HYDRA, these issues were addressed by removing arithmetic. This fundamentally breaks the mathematics and prevents the proxy-application from exploring numerical optimisations such as reduced precision. Further it only achieves similarity to that one HYDRA loop, but others are more significant contributors to runtime. The indirect writes were also removed from the primary CFD loop to ease collection of scaling performance data, but this reduces mini-HYDRA's similarity to HYDRA loops which perform the indirect writes with the CFD.

The restoration of indirect writes and the arithmetic leads to expected and specific differences in intra-node performance. Performance becomes memory-bound at different thread counts, but before that point both codes scale similarly. MG-CFD has increased sensitivity to L1 cache performance from increased register spilling, but sensitivity to the whole cache hierarchy remains similar. MG-CFD is further extended with MPI strong scaling with the OP2 DSL, and resulting parallel efficiency across a cluster of compute nodes is very similar.

Faced with the same issues as the original proxy-application developers, of specific performance differences to the target code, the next chapter will explore a different solution to the problem. This will be to use performance modelling to account for the difference, predicting the change in performance from the change in instruction content.

Chapter 5

Performance model of MG-CFD

The previous chapter identified that the proxy-application MG-CFD had a significantly different arithmetic intensity to the target loop compact-HYDRA. This led to MG-CFD becoming memory-bound at different core counts. It also complicated the task of benchmarking hardware, as the arithmetic intensity is sensitive to changes in ISA, and the effect of the difference is sensitive to the architecture. Thus it is insufficient to assume that the target loop of HYDRA would run at some fixed factor of MG-CFD's runtime. This is likely an issue faced by other proxy-applications of scientific modelling application with significant arithmetic components.

MG-CFD is designed to be very similar to HYDRA loops, operating on unstructured grid and performing CFD, differing only in the quantity and composition of floating-point arithmetic. This raises the question of whether the observed variance in arithmetic intensity can be quantitatively explained by this limited code difference, enabling runtime prediction of HYDRA loops without execution. This chapter seeks to answer that question, developing a performance model that predicts the *difference* in runtime that results from the *difference* in arithmetic relative to MG-CFD.

This chapter will begin with an investigation into the observed variance in IPC rates of MG-CFD and compact-HYDRA. It then designs a performance model to explain this, using approaches from mechanistic modelling, as it is necessary to understand how floating-point instructions are executed by the corresponding execution units within the target architecture. Model development adopts two goals - have minimal complexity, and require minimal technical information of the target architecture which may be novel with limited information. Complexity is progressively introduced only as simplifying assumptions are proven to prevent accurate prediction. This increases the likelihood that the model will be applicable to other architectures beyond those used for validation.

Table 5.1: Relative cost of double-precision FP DIV and SQRT instructions relative to MUL, in clock cycles

Arch	Scalar	SSE4	AVX2	AVX512
Broadwell	4-5x	4-5x	7-9x	-
Skylake	3-4x	3-4x	3-4x	6-7x
KNL	7x	5-6x	5-6x	5-6x

5.1 MG-CFD IPC investigation

As both MG-CFD and compact-HYDRA operate on the same unstructured grid, then the observed variation in IPC is likely due to the arithmetic differences between them. To accurately quantify this difference it is best to analyse the compiler-generated assembly code, as these will be the instructions actually executed. The main flux compute loops in both codes are simple one-dimensional loops over an unconditional sequence of arithmetic, so disassembling the object files and identifying the loop is mostly trivial. An exception is when the loop has been vectorised, as the compiler will generate an additional remainder loop with slightly different length, and potentially a second peel loop for data alignment. In this situation the detected loops within the disassembly are cross-referenced with the performance counter for executed instructions, selecting that assembly loop which is in agreement. In addition to identifying the instructions, the number of memory loads and stores are counted. This calculation is defined as the number of memory loads that are in addition to the minimum necessary to read in the unstructured grid data, which can be determined from source code analysis. To automate this loop identification and quantification process a lightweight Python tool has been written [43].

The instructions within the identified loop are categorised by throughput and type: low-throughput (LT) FP, high-throughput (HT) FP, and integer. Having a separate category for LT FP instructions is important as these have significant cost; for example one division or square root costs $3\times$ to $9\times$ more than a multiply instruction. A full breakdown is given in Table 5.1. Any significant change in the ratio of LT and HT FP instructions will alter overall IPC, so this must be quantified. Figure 5.1 presents the proportion of FP instructions that are LT for compact-HYDRA and MG-CFD. For compact-HYDRA this is approximately 5.5% under the AVX ISAs, and 1.2% lower at 4.3% under SSE. MG-CFD on CPU architectures exhibits the same pattern only doubled - approximately 11% under the AVX ISAs, and 2.6% lower at 8.4% under SSE4. The two codes diverge significantly when

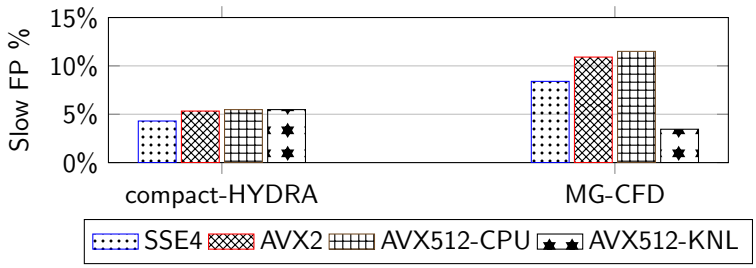


Figure 5.1: Proportion of FP instructions in flux loop that are relatively 'slow', meaning low throughput.

switching from Skylake to KNL under AVX-512; while compact-HYDRA has no change in instruction content, MG-CFD has a significant reduction in the proportion of LT FP, due to both a doubling of the number of HT instructions and a 38% reduction of LT instructions. Thus while both codes execute at a lower IPC on KNL, MG-CFD's reduction is partly offset by the increase in HT % FP. This explains the reduction of compact-HYDRA IPC relative to MG-CFD shown earlier in Figure 4.2.

5.2 Performance model development

The performance model for MG-CFD will be of the form of a mechanistic model focused on throughput through the processor FUs. It is expected that the compute bottleneck of both MG-CFD and compact-HYDRA will be the FP FUs, as these codes consist mostly of FP instructions and to my knowledge all processors are designed to decode instructions at a greater rate than they can pass through the FP FUs. Focus on the FUs also allows for the bottleneck residing on the load and store FUs, which may be the case if the code has a high quantity of register spilling. Being a parallel code, the bottleneck of parallel computation may shift to the achievable memory bandwidth, inter-process communication, or a combination of both. This is determined by the unstructured grid dataset, as the grid-specific indirect lookups determined by the ordering of the edges within arrays and numbering of the nodes constrain data throughput. Typically a publicly-available mesh is used by Rolls-Royce to assess unsecured systems, such as the NASA `Rotor 37` mesh of an axial compressor [49], so this can be used to directly measure the bounds of memory and communication performance. Only the compute performance of the commercially-sensitive loops within HYDRA, such as compact-HYDRA, need to be predicted. An "unsecured" system in this context does not mean it is known to be infected or vulnerable, instead it means a system which has not been explicitly approved and configured for securely hosting sensitive IP such as HYDRA.

5.2.1 Difference model

The first model will aim to predict the performance difference between MG-CFD and compact-HYDRA that results from the difference in instruction content. This model considers performance in terms of clock cycle consumption, named C_{mgcfd} for MG-CFD and $C_{compact}$ for compact-HYDRA. The model will focus on predicting the cycle consumption of a single loop iteration of compact-HYDRA, named $C_{g,compact}$, from $C_{g,mgcfd}$ of MG-CFD.

Once $C_{g,compact}$ has been estimated, then runtime prediction for thread count T is trivial given the number of iterations and clock frequency. The number of loop iterations performed is independent of hardware, as it is determined by the chosen mesh partitioner. It is also specific to each executing thread or process, as each can receive a different-sized partition, so the term $iters_t$ is introduced where t is the thread identifier. Compact-HYDRA is assumed to execute at the same clock speed as MG-CFD due to the high similarity between them, so the frequency term Hz_T is also independent of code. The cycle consumption of a particular thread of compact-HYDRA is calculated as:

$$C_{t,compact} = C_{g,compact} \cdot iters_t \tag{5.1}$$

Then the walltime of compact-HYDRA, $W_{compact}$, is calculated from the largest value of $C_{compact,t}$ across all threads:

$$W_{compact} = \frac{\max_{t=1}^{T} C_{compact,t}}{Hz_T} \tag{5.2}$$

Superscalar execution

To predict the resulting change in cycle consumption of the change in instruction content $\mathbf{\Delta I}$ requires a model of superscalar execution to reflect the complexity in modern architectures. A simple model is initially adopted, where each instruction category is scheduled to a single dedicated execution port, and each category is executed in parallel with, and independently of, other categories. The change in instruction content is assumed to be large enough such that when added to a kernel, the compiler is able to optimise the placement of individual instructions to maintain ILP, and so inter-instruction dependencies are ignored. Memory loads and stores are treated in a similar manner, with each mapping to a single dedicated port. For brevity, a memory load or store operation will be referred to as an instruction. There

are five instruction categories in this simple model: LT FP, HT FP, integer, loads, and stores. The change in instruction content is stored in the vector $\Delta\mathbf{I}$, where $\Delta\mathbf{I}_i$ is the difference in quantity of instructions in category i between MG-CFD and compact-HYDRA.

To begin calculating cycle consumption from instruction counts, the cost in cycles of each category is required. This is encoded in the vector $\mathbf{c}$, where $\mathbf{c}_i$ is the CPI estimate for category i (this estimation procedure is described later in Section 5.3). Now the core concept of the performance model can be introduced – port cycle consumption. This quantifies the cycle consumption of instructions in one category processed by the corresponding port; for brevity in model formulation and description, the port is treated as the consumer of clock cycles. Thus the term p_i can be introduced, the cycle consumption of port i. Given the known difference in instructions of category i, and the estimate of CPI, then p_i is calculated as:

$$p_i = (\Delta\mathbf{I}_i)\mathbf{c}_i \tag{5.3}$$

The predicted change in total cycle consumption between MG-CFD and compact-HYDRA is the maximum port cycle consumption, is formulated as:

$$\Delta C = \max_{i=1}^{5} p_i \tag{5.4}$$

Then $C_{compact}$ is given by:

$$C_{compact} = C_{mgcfd} - \Delta C \tag{5.5}$$

Contention

The model is extended further by considering hardware contention between different instruction categories. Figure 5.2 shows the portion of the Skylake microarchitecture pipeline related to instruction scheduling. It shows seven ports, four of these receiving integer or FP instructions, two receiving load instructions, and one receiving store instructions. Technically these ports receive μ-ops, but for simplicity these are referred to as instructions. On each clock cycle the scheduler can assign at most one instruction to each port. Ports 0 and 1 can receive both integer and FP instructions, revoking the prior assumption that each instruction category is scheduled to a dedicated port. Thus there is the possiblity of contention between different instruction categories.

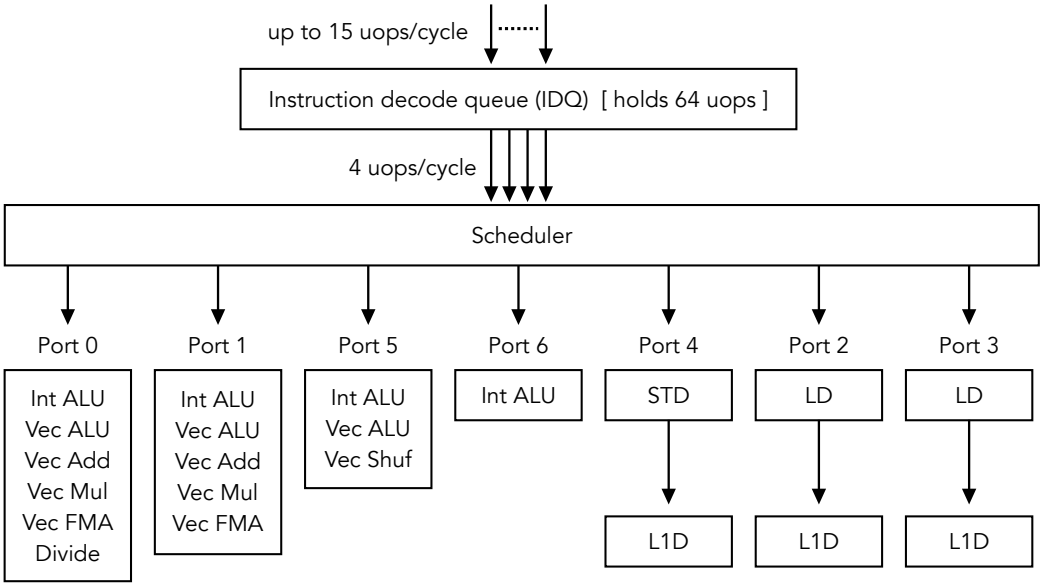


Figure 5.2: Instruction dispatch and issue stages of Xeon Skylake pipeline [28]

Accordingly, the model is extended to capture this contention, while maintaining a level of abstraction.

To implement this task, modelling makes two assumptions. These are made for the same reason, to avoid constructing the directed acyclic graph (DAG) of instruction execution from assembly, and to avoid the additional cost of repeatedly identifying the critical path in the CPI estimation process. The validity of these assumptions will be determined by the accuracy of the model predictions. A low error justifies excluding the additional modelling of inter-instruction dependencies.

The first assumption is that while an FU is occupied then so is its resident port, blocking all other FUs on it. The second assumption is of an ideal instruction scheduler that can schedule in bulk the cycle consumptions of all instructions, scheduling first those instructions with the fewest compatible ports and with the objective to minimise the maximum clock cycle consumption across the ports. This process is visualised in Figure 5.3, configured for the Skylake architecture. This figure also states the instruction categories to be used by the model, which contains two more instruction categories than the earlier discussion regarding assembly categorisation. These have been created to improve the fit of model to architecture, and to better discriminate between sets of instructions that may execute at different throughputs. The `FP shuf` category contains those instructions that map specifically to the `Vec Shuf` FU on port 5, unlike most other FP instructions that map to ports 0 and 1.

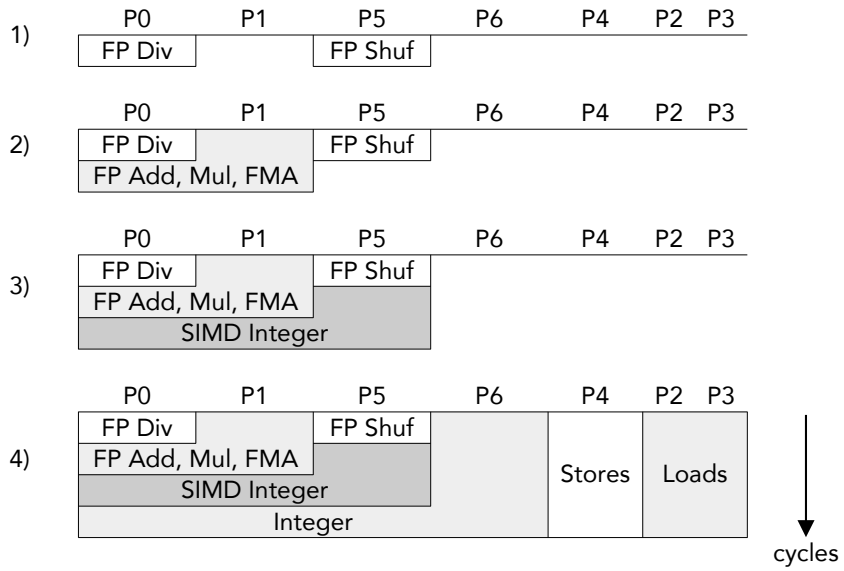


Figure 5.3: Ideal scheduling model of instructions to execution ports within Skylake. Instructions with fewest compatible ports are scheduled first.

The `SIMD Integer` category is created from the more general integer category as it maps to one less port, and being vectorised may have a lower throughput than its serial counterpart.

This model seeks to predict the change in performance that results from the change in instructions from MG-CFD to match compact-HYDRA. The first stage is to predict how those modified instructions were scheduled to ports, using the previously stated assumptions. This produces an allocation matrix $\mathbf{A}$, where $\mathbf{A}_{p,i}$ is an allocation of some or all instructions of category i to port p, with the following structure:

$$\mathbf{A} = \begin{array}{c} \begin{array}{ccccccc} \text{DIV} & \text{FP Shuf} & \text{Load} & \text{Store} & \text{FP} & \text{SIMD Int} & \text{Int} \end{array} \\ \begin{bmatrix} d & - & - & - & f_2 & v_3 & i_4 \\ - & - & - & - & f_1 & v_2 & i_3 \\ - & - & - & - & - & v_1 & i_2 \\ - & p & - & - & - & - & i_1 \\ - & - & l_2 & - & - & - & - \\ - & - & l_1 & - & - & - & - \\ - & - & - & s & - & - & - \end{bmatrix} \end{array} \begin{array}{l} \\ \text{P0} \\ \text{P1} \\ \text{P5} \\ \text{P6} \\ \text{P2} \\ \text{P3} \\ \text{P4} \end{array}$$

The ideal instruction scheduler fills this matrix from left to right, prioritising those instruction categories with the fewest available ports, seeking to minimise the maximum port cycle consumption. Then cycle consumption of port i is the vector dot product between its allocated instructions and their respective CPI estimates (i.e. different categories with a port are summed together, not treated as parallel):

$$p_i = \mathbf{A}_i \cdot \mathbf{c} \tag{5.6}$$

Then the observed change in clock cycle consumption is equal to the greatest cycle consumption of a single port. Where N_p is the number of ports:

$$\Delta C = \max_{i=1}^{N_p}[p_i] \tag{5.7}$$

A flaw in this formulation is the assumption that any change in any instruction category will lead to an observed change in clock cycle consumption determined by properties of just that category. Trivial counterexamples show this to be false. Consider a kernel dominated by LT FP instructions executing in the example Skylake architecture, as shown in Figure 5.2. Performance would be bound by throughput of these instructions through execution port 0; any HT FP instructions would be modelled as executing through port 1, as this would be under utilised. Removal of a single HT FP instruction would have no measurable effect on runtime, as performance is bound by port 0 throughput. Consider another kernel which is dominated by HT FP instruction, rather than LT. If a single LT instruction was removed, a reduction in runtime is expected but will be determined by properties of both LT and HT instructions, specifically the cost of the former and the scheduling of the latter.

The solution is to model the total cycle consumption of MG-CFD, then of compact-HYDRA, and use their difference as the prediction of ΔC. Note that these predictions of total cycle consumption are not required to be accurate, only their difference is required to be accurate. For MG-CFD, its total cycle consumption is modelled much like previously described, except to use its total instruction content rather than difference:

$$C_{model,mgcfd} = \max_{p=1}^{N_p}[\mathbf{A}_{mgcfd,p} \cdot \mathbf{c}] \tag{5.8}$$

Similarly for compact-HYDRA:

$$C_{model,compact} = \max_{p=1}^{N_p}[\mathbf{A}_{compact,p} \cdot \mathbf{c}] \tag{5.9}$$

Then the revised calculation of ΔC becomes:

$$\Delta C = C_{model,mgcfd} - C_{model,compact} \tag{5.10}$$

The final calculation for $C_{compact}$ is unchanged:

$$C_{compact} = C_{mgcfd} - \Delta C \tag{5.11}$$

5.3 CPI estimation

Critical to calculating the cycle consumption of the aggregate instructions is having an accurate measurement of their individual cost of cycles, CPI. This is treated as an unknown that must be estimated through benchmarking with MG-CFD, for three reasons:

- For novel architectures these values may not be public knowledge, unlike for older architectures which have been thoroughly assessed [21]
- FUs for expensive operations can be pipelined, such as FP division, allowing for their full cost to be masked by ILP. This ILP is partly a property of the code being executed, specifically of the data dependencies between instructions, so the achieved CPI values will have some specificity to the code being executed. It is assumed that the class of CFD codes have similar ILP, meaning MG-CFD's CPI measurements will translate to compact-HYDRA and to other kernels in HYDRA.

- The model being an abstraction of FU scheduling may require the flexibility to fit to the observed data

Collecting varied performance data is critical to avoid overfitting and thus poor accuracy when predicting HYDRA kernels. This requires distinct variants of the MG-CFD flux routine each with different proportions of FP instructions. To create these, the combinatorial enabling of four arithmetic optimisations missed by the compiler produces approximately eleven distinct kernels, with the exact number depending on the compiler used. These variants contain different quantities of division, square-root, all other FP operations, and integer operations, while producing the same numerical result. Additional variants are created by also using different compilers, in this case the Intel and GNU C++ compilers. Between these two compilers and arithmetic optimisations, twenty distinct variants of the MG-CFD flux routine are created, but they span a relatively narrow range with the difference between most and least expensive variant being approximately 30 instructions (17% of total). To increase this range an additional kernel is created, derived from the flux routine but with $\approx 50\%$ arithmetic instructions removed. This resulting kernel does not correctly implement the flux accumulation, requiring another correct variant to be executed to maintain solver convergence, however it enables the range to be extended greatly. This larger range produces better CPI estimates that greatly reduce prediction error of $C_{g,compact}$.

An appropriate optimisation technique is required to fit the execution model to the MG-CFD performance data. Note that the use of *max* across multiple linear equations in Equation 5.7 renders the function for C nondifferentiable despite the individual equations for p_i being differentiable. This means that direct application of a numerical minimisation method is unlikely to find the global optimal set of CPI values. A stochastic technique ideally suited to this problem is basin-hopping [63]. Like simulated annealing, it 'hops' through the parameter space to escape local minima, but unlike simulated annealing it then performs local gradient minimisation at each hop. Domain knowledge is applied to the parameter space to constrain CPI estimates to have a minimum of 1.0. This technique is implemented in the Python `SciPy` library within the `optimize` module.

Table 5.2: Single-thread compact-HYDRA runtime prediction errors of the three described models.

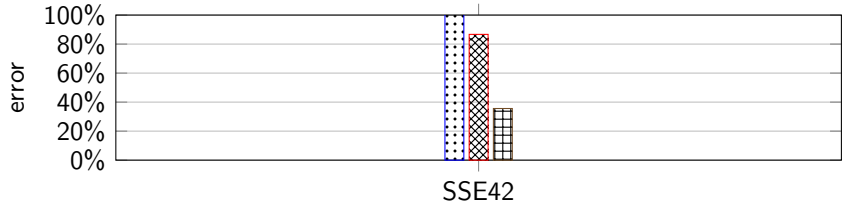
Model	mean (%)	SD (%)	max (%)
Equal IPC	63.2	50.4	166.0
Constant R_c	26.8	30.1	88.4
Difference model	13.9	9.6	35.6

5.4 Model validation

To justify the development of this model, its predictive accuracy will be contrasted with two naive models. The constant-R_c model assumes that the ratio of $C_{g,compact}$ to $C_{g,mgcfd}$, defined as R_c, is constant across architectures and ISAs (ignoring bounds imposed by memory performance). The equal-IPC model assumes that both MG-CFD and compact-HYDRA execute with the same rate of IPC, so cycle consumption is directly proportional to the number of instructions executed.

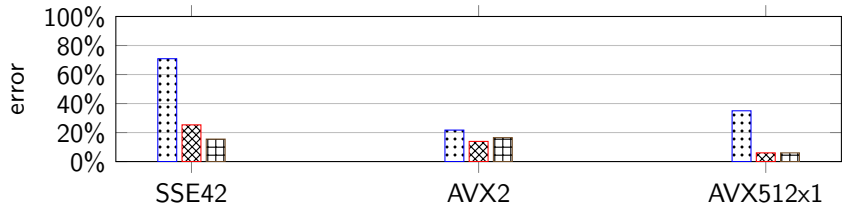
Each model is evaluated by its ability to accurately predict single-threaded cycle consumption of compact-HYDRA. Assessment is performed across a range of different CPU architectures and instruction sets, spanning nine years of development, that will test the robustness of each model. These architecture are Intel's Xeon's Westmere, Broadwell, Skylake, Cascade Lake, and KNL. Full hardware details are provided in Table 3.2 in Chapter 3. The ISAs are AVX512, AVX2 and SSE42. AVX and SSE41 are ignored as their differences to AVX2 and SSE42 are very minor. R_c is measured for Broadwell AVX2 and then assumed to be constant for all other combinations of systems and ISAs.

Figure 5.4 presents accuracies of each model, with accompanying statistics listed in Table 5.2. The equal-IPC model produces predictions of poor accuracy; of the twelve predictions, seven have error exceeding 40%, and only one error is below 20%. This is clear evidence that MG-CFD and compact-HYDRA can, and often do, execute at significantly different overall IPC rates. The constant-R_c model performs better, generating seven predictions with below 20% error. Of particular note is that for architectures that have a high degree of similarity to that used for calibrating R_c, errors are consistently low. Specifically, with Xeon Broadwell used for calibration, predictions of its successors Skylake and Cascade Lake show low errors of mean 15.3%. Skylake can be considered an architectural tweak of Broadwell, exploiting a greater transistor budget to widen the superscalar pipeline and add new instructions, but not make large fundamental changes. Cascade Lake is

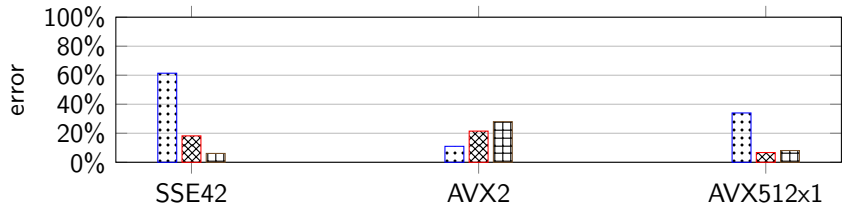


(a) Xeon Westmere

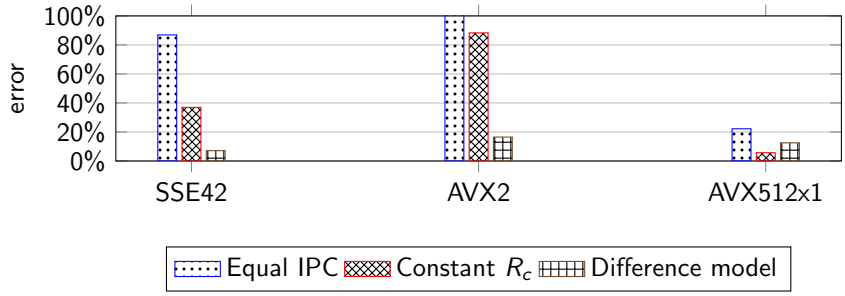
(b) Xeon Broadwell



(c) Xeon Skylake



(d) Xeon Cascade Lake



(e) Xeon KNL

Figure 5.4: Model prediction errors of single-threaded compact-HYDRA cycle consumption.

simply a transistor node shrink of Skylake, with unchanged architecture and higher clock frequency, so model predictions for it are expected to closely match Skylake. However, predictions for the significantly different architectures Westmere and KNL have high errors; Westmere prediction has 86.7% error, and KNL predictions have mean error 43.7% and maximum error 88.3%. The Westmere architecture predates Broadwell by two generations (with Sandy Bridge between them), designed during Intel's transition away from the Pentium 4 architecture to Sandy Bridge which required fundamental architectural rewrites; KNL architecture is derived from the low-power Atom architecture, with greatly reduced superscalar and out-of-order capabilities relative to Xeon CPUs. These results indicate that when the target architecture is significantly different to that used for calibration, the constant-R_c model becomes unreliable. In contrast to the previous models, the difference model does produce predictions of low error across all tested architectures, with mean error 13.0% and maximum error 39.6%. The variance of the errors is also low, with a standard deviation of 9.6%, indicating that the model can be relied upon to accurately predict performance on new architectures.

5.5 Predicting strong scaling

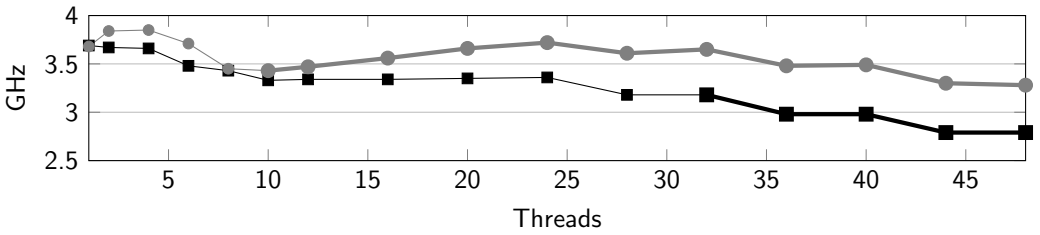
5.5.1 Memory benchmarking

In order to predict compute performance across multiple cores with a single node, it is necessary to account for the interaction between the kernel code and memory hierarchy performance. As more processor cores are utilised in a multicore system, demand on the shared resource of main memory increases. Depending on the arithmetic intensity of the kernel, its scaling performance may be ultimately bounded by data throughput of the hierarchy, when memory demand exceeds its capacity to deliver. Although a memory system is specified with a maximum theoretical read/write throughput, this is unlikely to be achieved. A kernel that performs constant-stride memory accesses will trigger optimal performance from the hardware prefetcher, and fully utilise all fetched cache lines, will achieve nearest to the theoretical bandwidth. For example, vendor-optimised versions of the STREAM benchmark achieve 80% of the maximum theoretical bandwidth [16]. The irregular memory accesses of unstructured grid codes greatly reduce achievable bandwidth, by hindering the hardware prefetcher, and not fully utilising fetched cache lines.

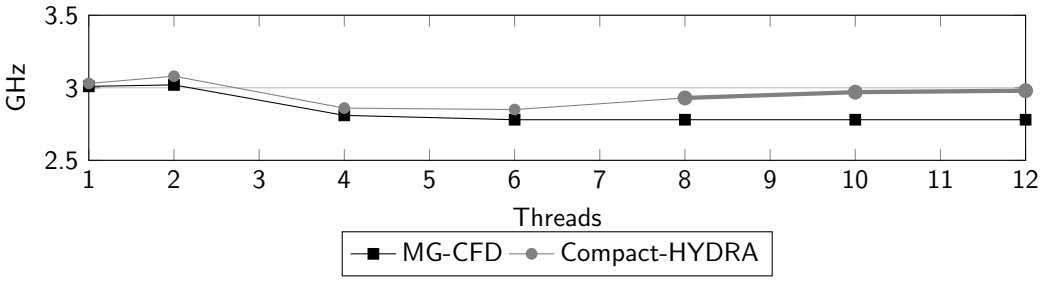
To measure the maximum bandwidth achievable by unstructured grid codes, a new data throughput (DT) kernel is added to MG-CFD that encodes the same memory access pattern (determined by the input grid), but with minimal arithmetic

computation to be inherently memory-bound. It is not feasible for this kernel to perform no arithmetic at all, as then any sensible compiler optimiser would remove the entire kernel. The quantity of arithmetic remaining is expected to be less than that of any other possible kernel that performs useful work on the same dataset, thus establishing a bound on achievable memory performance; for example, when compiled to the AVX2 ISA, the DT kernel performs just 13 floating-point operations across 20 memory loads and 10 memory stores. With such a low flop/byte ratio, it is expected that this kernel will establish the maximum speed at which any given system can traverse through an unstructured grid, thus establishing the memory bound of codes operating on that grid. To confirm the earlier claim that unstructured grid codes have a lower achievable bandwidth than STREAM, execution of this DT kernel on Xeon CPU systems achieves between 52% and 62% of the bandwidth achieved by STREAM. The empirical memory bound is incorporated into the model using a simple classification. Predicted performance of compact-HYDRA at thread count t is classified as being memory-bound if predicted compute runtime $T_{t,compact}$ is less than the measured $T_{t,DT}$; if true, then predicted compact-HYDRA runtime will be raised to $T_{t,DT}$. The term t_b is introduced, which is the lowest thread count t at which compact-HYDRA is expected to be memory-bound on a particular system.

Another dynamic property of a multicore system to consider is the variable clock frequency of its processor cores. Modern processors operate at a variable and clock frequency, controlled dynamically in real-time by processor logic according to circuitry temperature and power draw, primarily determined by the particular instruction mix and throughput of the executing code. The variance in clock frequency can be significant; for example in the Xeon Broadwell node, single core execution of MG-CFD operates at 3.19 GHz, which drops by 34.8% to 2.08 GHz when all cores are utilised. Such large variance can introduce large errors into the model predictions, particularly when predicting compute-bound runtime, so it is essential to know the frequency that compact-HYDRA will execute at at any particular thread count. This is expected to be equal to MG-CFD at any identical thread count due to the similarity between the codes. To confirm this, scaling frequency of both codes is measured on Xeon Westmere and Cascade Lake, distinctly different architectures. Xeons Broadwell and Skylake behave similarly to Cascade Lake, and KNL has little variance due to a lower base clock, so these are excluded. Presented in Figure 5.5, both codes do execute at similar clock frequencies when compute-bound, with the difference not exceeding 6%. It is not necessary that frequencies match when execution is memory-bound, as predicted performance will be determined by the memory hierarchy not core architecture and so the empirical runtime of the DT



(a) Xeon Cascade Lake



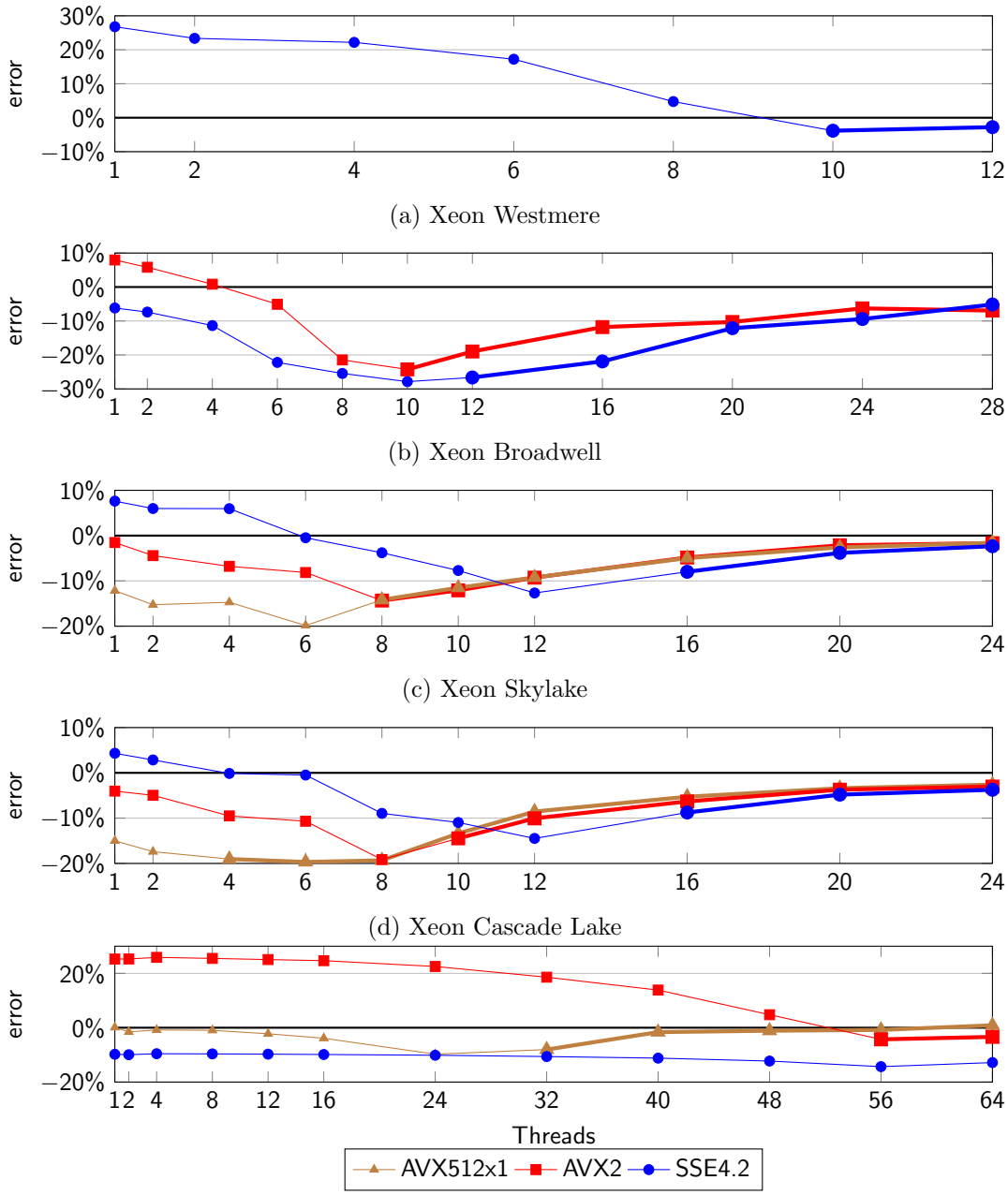
(b) Xeon Westmere

Figure 5.5: Relationship between multicore load and observed turbo GHz. Bold line indicates when code is memory-bound, thin line when compute-bound. When both codes are compute-bound they operate at similar clock speed, important for model accuracy.

kernel will be used.

Figure 5.6 presents model accuracies of multicore predictions for compact-HYDRA grind time, for the same systems as used for single-core model validation. This error can be negative or positive, indicating when the model prediction is below or exceeds actual runtime (respectively, to better present the effect of thread count on model prediction. The relationship between prediction accuracy of compute-bound runtime and thread count is similar on each system, of the error falling in value as the thread count approaches t_b of each respective system. Any positive error will reduce towards zero, then become an increasingly negative error until predicted performance is memory-bound. This is attributed to $C_{g,compact}$ increasing as compact-HYDRA approaches t_b, a behaviour not considered by the model which assumes constant cycle consumption for compute-bound execution. For most systems this behaviour begins immediately from $t = 1$; only on KNL is model error constant across a range of thread counts, due to t_b being much higher than for the other systems; this is evidence that when compact-HYDRA is far from being memory-bound then $C_{g,compact}$ is constant.

The observed trend of $C_{g,compact}$ increasing near its memory-bound is explored further, as for some configurations it leads to relatively large changes in error; for example on Xeon CPU systems it often leads to a 15% increase in absolute error,



(a) Xeon Westmere

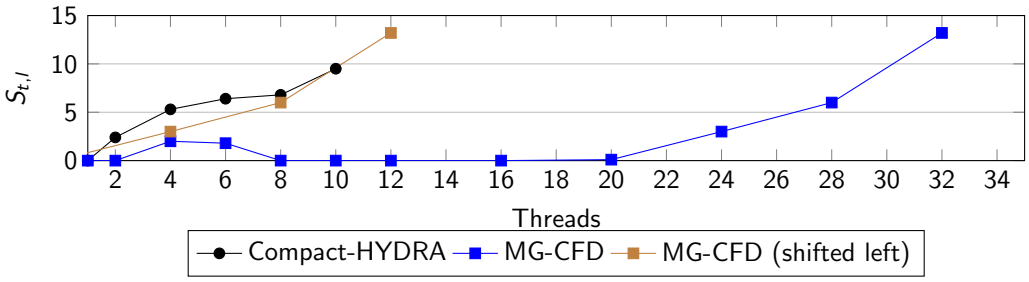
(b) Xeon Broadwell

(c) Xeon Skylake

(d) Xeon Cascade Lake

(e) Xeon KNL

Figure 5.6: Model prediction errors of compact-HYDRA strong scaling. A negative error represents an under-prediction of actual performance. Bold line indicates when predicted performance is memory-bound, thin line when compute-bound. Cascade Lake sequence clipped at 24 threads for brevity.



(a) Xeon Cascade Lake

Figure 5.7: Relationship between thread count and stalled cycles for compact-HYDRA and MG-CFD, at compute-bound thread counts. Each kernel encounters increasing penalty of similar size when approaching t_b, made clear by shifting MG-CFD datapoints.

resulting in a near 30% error on Xeon Broadwell. This increase in cycles is attributed to stalling of execution from saturation of memory throughput, as there is no change in the quantity of computation nor change in the architecture as a result of increased thread count. For clarity the term $C_{g,t}$ is introduced, which is the clock cycle consumption of one loop iteration of either compact-HYDRA or MG-CFD on Then the number of stalled cycles $S_{g,t}$ at thread count t is defined as:

$$S_{g,t} = C_{g,t} - C_g \tag{5.12}$$

Further investigation is performed on the Xeon Cascade Lake system, as this is the only system on which MG-CFD becomes memory-bound. $S_{g,t}$ for both compact-HYDRA and MG-CFD is plotted for $t \leq t_b$ in Figure 5.7. Both codes exhibit a similar behaviour of increasing $S_{g,t}$, both in terms of magnitude and the range of thread counts, ony differing by the thread count at which the behaviour begins. This is made clear by shifting the datapoints of MG-CFD to overlay those of compact-HYDRA. This similarity indicates that the effect is independent of kernel size, and that its importance diminishes with increasing kernel size. It also indicates that the effect could be measured with a sufficiently-low flop/byte ratio variant of MG-CFD (to ensure it encounted memory-bound on target system), and then directly added to current model predictions.

Returning to Figure 5.6, another model divergence is that for memory-bound thread counts soon following t_b, the model often underpredicts compact-HYDRA runtime; as thread count continues to increase, the error converges to a near-zero underprediction with increasing thread count. This is despite having empirically

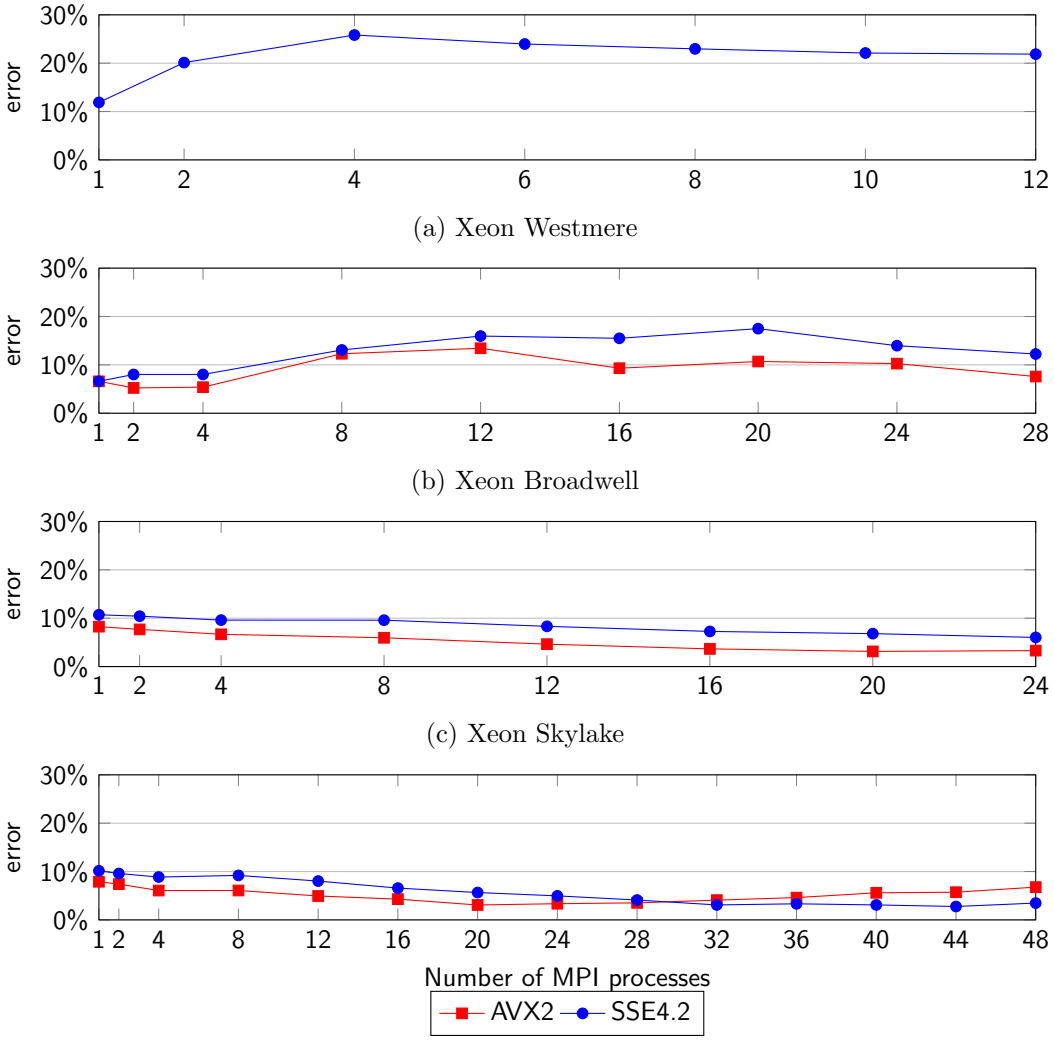
measured the runtime of the data movement. This indicates that the core architecture is not able to fully overlap computation with data movement, but that this effect diminishes as thread count further increases and compute becomes a lesser percentage of runtime. As with the stalled cycles during compute-bound execution, this imperfect overlap could be measured and used to adjust the model prediction, but considered relatively unimportant for further application of the model.

5.5.2 Predicting performance of HYDRA

Having validated the predictive ability of MG-CFD and performance model, attention is directed to the most significant HYDRA kernel, `vflux`. This is the single most expensive loop in HYDRA; for 28 MPI processes on Xeon Broadwell, it accounts for 35.8% of the walltime. Accordingly its arithmetic intensity is several times that of MG-CFD, posing a significant challenge to the projection model. In contrast, its data access pattern is very similar to that of compact-HYDRA, performing the same single loop over edges, and only differing significantly in the quantity of data associated with each node (cell).

For this prediction task a different dataset is used that is typical for a HYDRA workload, the NASA `Rotor 37` mesh of an axial compressor rotor [49]. This contains approximately 8.1 M nodes and 24 M edges, with an additional three MG meshes that results in a total count of approximately 15.7 M nodes and 53 M edges. Thus, this is a significant size for assessing single-node performance. Further, as HYDRA implements strong scaling, enabled by the invocation of mesh partitioning routines, then each individual MPI process will be operating on a different mesh structure. This is in contrast to MG-CFD being executed in a weak scaling manner on a mesh 26.4× smaller.

As performed for the compact-HYDRA predictions, the assembly code of the `vflux` loop is extracted from the compiler-generated object file and its constituent instructions categorised. The set of MG-CFD variants are executed on each target system, providing empirical data for estimation of CPI rates. The projection model is applied to provide an estimate of $C_{g,vflux}$, the cycle consumption of a single `vflux` loop iteration. MG-CFD also measures clock speed, allowing $C_{g,vflux}$ to be converted into grind time, the runtime of a single loop iteration. The grind time is passed into a pre-existing performance model of HYDRA, which combines it with knowledge of mesh partitioning and a function call trace to produce a prediction of total compute runtime for each HYDRA kernel [9]. For more details on this pre-existing HYDRA performance model, not to be confused with the novel MG-CFD performance model, refer back to Section 3.2.1 in Chapter 3.



(a) Xeon Westmere

(b) Xeon Broadwell

(c) Xeon Skylake

(d) Xeon Cascade Lake

Figure 5.8: Model error of predicted HYDRA vflux() compute strong scaling.

Table 5.3: Model error statistics of predicted `vflux` compute strong scaling.

System	ISA	mean (%)	SD (%)	max (%)
Westmere	SSE4.2	21.2	4.5	25.8
Broadwell	AVX2	9.0	3.0	13.4
Broadwell	SSE42	12.3	3.9	17.5
Skylake	AVX2	5.4	2.0	8.3
Skylake	SSE4.2	8.6	1.8	10.7
Cascade Lake	AVX2	5.2	1.5	7.9
Cascade Lake	SSE4.2	5.9	2.8	10.1
Overall		8.8	5.5	25.8

Predictions are made of strong scaling of `vflux` compute; time spent halted on MPI synchronisation routines are ignored as this is determined mostly by mesh workload imbalance rather than CFD calculations. Let $V_{l,r}$ be the time that MPI rank r spends within the compute loop of `vflux` on MG level l. Then model error is calculated as the sum of *absolute* error of predicted $V_{l,r}$ across ranks and levels, rather than real (signed) error, to ensure that a mix of under- and over-predictions do not cancel out and give the illusion of a highly-accurate model:

$$\text{model error } \% = \frac{\sum_{l=1}^{levels} \sum_{r=1}^{ranks} |\text{error}_{l,r}|}{\sum_{l=1}^{levels} \sum_{r=1}^{ranks} V_{l,r}} \tag{5.13}$$

This model error is plotted for several Xeon CPU systems, shown in Figure 5.8, and accompanying statistics listed in Table 5.3. Predictions for the AVX-512 ISA are missing due to a segfault bug within the Intel compiler, that prevents MG-CFD from being compiled with aggressive floating-point optimisations disabled when targeting AVX-512, necessary to match HYDRA's compilation configuration; this bug did not affect data collection during the prior model validation as aggressive floating-point optimisations were enabled, not disabled. Across all systems and rank counts, mean model error is 8.8% with a standard deviation of 5.5% and maximum error 25.8%. These errors are lower than when predicting compact-HYDRA performance, despite the greater differences between MG-CFD and HYDRA's vflux routine, and also that vflux is approximately 3-4× more expensive than MG-CFD. This could be attributed to the model assumptions holding better for a kernel with more assembly instructions, specifically the assumption that MG-CFD and the target kernel contain sufficient instructions for the processor instruction scheduler to maintain a similar level of ILP, allowing any inter-instruction dependences to be ignored.

5.6 Vectorising unstructured grid compute

One of the primary uses for mini-applications is the assessment of optimisations and novel architectures. One particular feature of interest are floating-point vector units, increasingly used by vendors to highlight the peak compute capability of their new systems. Indeed, vector units do have the potential to greatly reduce runtime of simulation codes, but obtaining good performance from them is not always simple. This is a particular problem for unstructured grid codes, where their indirect memory accesses prevent the naive use of the vector units. MG-CFD is used to assess two schemes for safely vectorising unstructured grid compute, and then with the new performance model the achieved performanced is analysed.

5.6.1 Conflict avoidance

To vectorise the indirect writes of unstructured grid compute, it is necessary to ensure that either (a) conflicting writes do not occur or (b) that they are handled safely. A conflict here means that at the end of a vectorised iteration, two or more independent writes are destined for the same memory address. Two strategies exist to avoid this. One is to vectorise the compute but serialise the indirect memory writes, as used by the OP2 library [41]. This requires the programmer to manually and sequentially unpack each vector register, constraining processor pipeline throughput. It may also be necessary to manually pack if no `gather` instructions are present in the architecture. A variant of the serial-write strategy is to use specialised hardware to detect conflicts at runtime, generating a vector mask that allows safe writes to be vectorised, then use the mask inverse to serialise the conflicting writes. An example implementation can be found within the AVX-512-CD instructions, implemented in KNL, Xeon Skylake and Xeon Cascade Lake architectures. While requiring no additional developer effort, this solution still results in the compiler generating longer loops to accommodate the new masked-write loops.

The second strategy is to reorder the loop iterations to guarantee that within any vector write there are no conflicts, which allows the compiler and processor architecture to optimise the memory writes. This is achieved by treating the mesh as a graph and colouring the edges by their connectivity, then packing each vector with edges of equal colour. In MG-CFD a greedy sequential colouring algorithm is used, where the uncoloured edges are processed in descending order of their degree of coloured edges. Then a greedy reordering algorithm is applied, packing each vector-sized block with edges of colour equal to the first unpacked edge. Note that this reordering disturbs the spatial locality in proportion to the number of colours

required, which does increase through the multigrid. This scheme contrasts with one that reorders *all* edges of a colour together, which would further reduce spatial locality. Colouring and reordering is performed once during initial data load, and is inexpensive relative to MG-CFD's compute. If during packing of a vector block there are insufficient edges of equal colour to fully pack the block, then those edges are processed serially in a remainder loop.

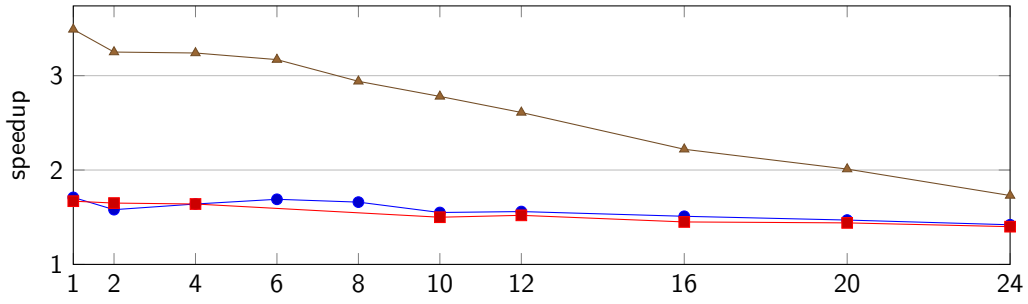
5.6.2 Vectorisation performance

The two described software-based schemes are implemented in MG-CFD. The `manual` scheme is that which serialises the indirect writes, and the `colour` scheme is that which colours and reorders the edges. Vectorised code is generated by an auto-vectoriser, rather than hand-coding intrinsics; hand-coding would be inherently non-portable, negating the rapid benchmarking ability of MG-CFD. To ensure that both the flux computation kernel and DT kernel are performing the same data movement, it is necessary that the auto-vectoriser maps each to the same vector width. Any sensible compiler would not do this, as each loop has a very different flop/byte ratio. To force equal width the OpenMP SIMD API is used, by inserting `omp simd simdlen(INT)` pragmas.

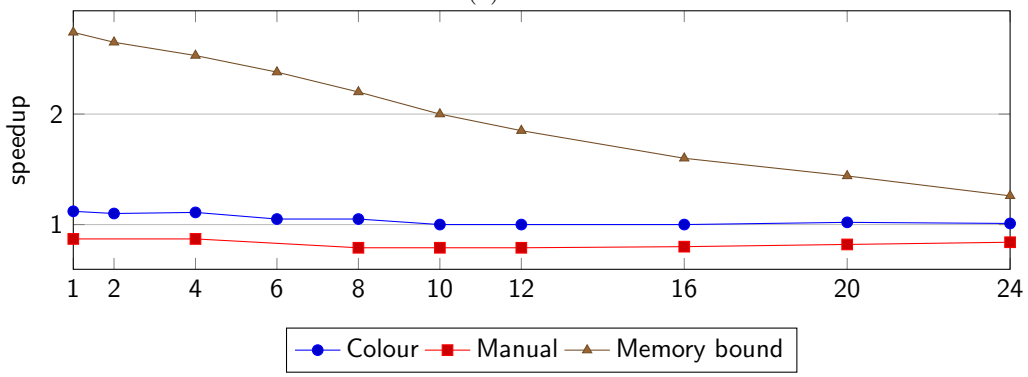
Performance data is collected on the Xeon Skylake node (details in Table 3.2), using the Intel compiler, and targeting both AVX512 and AVX2 ISAs. This particular Skylake architecture contains two non-vectorised FP units, two 256-bit vector units supporting AVX2, but only one 512-bit vector unit supporting AVX512. Then it could be expected with either ISA that optimal MG-CFD vectorised performance would achieve a 4× speedup for double-precision (64-bit) floating-point compute.

Speedups are measured and calculated across the full range of thread counts, using the DT kernel performance to bound the maximum achievable speedup by memory performance. These are shown in Figure 5.9. Targeting AVX2, each scheme achieves a similar speedup at each thread count, ranging from 1.7× single-threaded down to 1.4× at 24 threads. With AVX512 any speedup is negligible, and the manual scheme results in a small slowdown of approximately 1.2×. These are far from the desired 4× speedup, and at low thread counts are far from the memory-bound.

There are several confounding factors that each reduce the achievable speedup. To better understand these, the single-threaded performance will be investigated. One factor is the clock frequency that the processor operates at during execution. This responds to heat and power consumption, which increases with wider vector units and typically leads to lower frequencies. Table 5.10 shows the clock frequency that each code configuration executes at. Vectorised AVX2 frequency changes little



(a) AVX2



(b) AVX512

Figure 5.9: Vectorisation speedups with two conflict-avoidance schemes on Xeon Skylake. Also shown is maximum speedup permitted by achievable memory performance.

Figure 5.10: MG-CFD GHz with and without vectorisation. Only with AVX512 does vectorisation reduce GHz

ISA	Serial	Colour	Manual
AVX2	2.99	2.89	2.89 (97%)
AVX512	2.99	1.8	1.8 (60%)

Figure 5.11: Changes to floating-point quantity and throughput in vectorised MG-CFD. Increased quantity or lower throughput reduces achievable speedup

Scheme	Fast FP	Fast FP IPC	Slow FP IPC	**FP cycles/iter**
None	112	0.60	0.16	269
Manual	152 (1.36×)	0.33 (54%)	0.10 (64%)	**595 (2.21×)**
Colour	112	0.29 (48%)	0.05 (35%)	**626 (2.32×)**

AVX2

at 3.3% slower, but vectorised AVX512 clock speed is 40% slower. This is another near-50% reduction in the achievable speedup of AVX512 vectorisation, which when added to the loss from port fusion leaves a peak theoretical speedup of just 2×, half that of AVX2.

To complete the identification of confounding factors, the disassembly analysis tool is applied to extract the instructions constituting the main vectorised loop, and the MG-CFD performance model is used to estimate the CPI rates of the instruction classes (and by extension their inverse IPC rates). These numbers are presented in Table 5.11. The additional serialised 'increment' loops performed by the manual conflict avoidance significantly increase the number of fast FP instructions, by 1.36× with AVX2. Both schemes exhibit reduced floating-point throughput; fast FP is reduced to approximately 50% for both schemes, and slow FP reduced to 64% and 35% for the manual and colour schemes respectively. If there is any potential for improving the speedup obtained by either scheme, it is in increasing these throughputs to match unvectorised floating-point IPC, and should be the focus of further investigation. The net result on the cost of the floating-point workload per loop iteration is a 2.21× increase with the manual scheme, and 2.32× with the colour scheme.

5.7 Summary

This chapter identifies the issues that can arise when using a proxy-application to make inferences of the expected performance of a target application. Relatively minor differences in the code content can result in significant differences in performance when assessing new architectures. With the use of MG-CFD to directly infer performance of compact-HYDRA, this issue manifested in two ways: (a) different overall IPC, and (b) different memory-bounds. These manifestations would greatly impair the ability to predict speedup of a new multicore nore and/or new architecture.

It is then shown how the similarity between such a proxy-application and its target enables the use of a relatively simple performance model to bridge the divide, to account for the differences in observed performance that result from the few static differences between them. With MG-CFD, this is a simple execution model of instruction throughput that can ignore inter-instruction dependencies. This approach is validated by the accurate prediction of compute performance of HYDRA's `vflux` loop, with mean error 8.8%, despite being approximately 3-4x more expensive and operating on different mesh structures.

Finally, the model is used to explore the efficacy of vectorising unstructured grid compute. Of the theoretical maximum speedup of 4×, only 1.7× was achieved. This is attributed not to hitting a memory-bound, but to throughput of vectorised FP instructions being significantly less than of their serial counterparts.

Chapter 6

Conclusion

This research has explored an issue with the use of proxy-applications, of the reliability of the performance assessments they make regarding their target application on new architectures or instructions sets (ISAs). Focusing on one particular pairing, the production CFD simulation code HYDRA and its proxy-application MG-CFD, significant discrepancies were identified between their performances on certain hardware and ISA combinations. This introduces uncertainty into the accuracy of HYDRA performance predictions produced by MG-CFD.

Recognising that the code differences between key HYDRA loops and MG-CFD are few, a performance model is proposed that predicts the difference in runtime between the two codes that results from those code-level differences. Specifically, this model predicts the effect on MG-CFD's runtime of adding or removing a bulk quantity of assembly instructions to the compiled MG-CFD's flux computation kernel, where that addition or removal would result in a kernel matching the desired target kernel. This is a high-level model, considering throughputs of classes of instructions (measured empirically), and ignores inter-instruction dependencies. This model accurately predicts performance of two distinct HYDRA kernels, one with much lower arithmetic intensity and a second with much higher, with mean errors of 13.9% and 8.8% respectively. This research hopefully serves as motivation to other proxy-application developers, to consider constructing an accompanying performance model to account for those static differences, providing confidence in performance assessments made by the proxy-application.

Further, this model provides a framework for evaluating more elaborate code changes, such as a transformation into a vectorised loop. Analysing specifically auto-vectorisation on a Xeon Skylake node, poor speedups were obtained of approximately 1.7× with AVX2 and negligible change with AVX512. Model IPC estimation identified

that floating-point throughput had approximately halved, independent of the mesh ordering; this indicates a computational bottleneck within the processor pipeline that limits the speedup obtainable by unstructured grid compute. This analysis allows expectations of achievable speedup of target application to be set realistically, away from the ideal 4×.

6.1 Limitations

A key limitation of the applicability of the performance model is the need for disassembly of the target kernel, eg `vflux`, generated to target the potentially-unsecure system of interest. Obtaining this assembly would require access to the vendors compiler and executing it locally to the source code, within the secure firewall where the source code resides. This would be a problem if the vendor requires that the compiler be executed in their system, possibly because they consider the disassembly to be sensitive because it discloses architectural or compiler details under a NDA. I am unaware of technical issues that would prevent the generation of disassembly, as all typical compilers allow for specific instruction sets to be targeted regardless of what system the compiler is executing on. The four compilers considered typical are GNU, Intel, Clang, and Cray.

Another limitation is the need for a performance counter library to exist and function on the vendor system of interest. This is in order to precisely instrument two particular loop in MG-CFD, that of flux computation and that of DT, and measure their instruction and clock cycle consumption. Without such a library, the only easy means to collect performance counter data is with the Linux utility `perf`, but this monitors across the entire binary. Such a solution would require modifying MG-CFD execution such that one particular loop comprises near-100% of runtime.

Finally, the performance model has only been validated on CPU processors, albeit including many-core. GPGPUs are increasingly important in achieving high-performance computing, particularly in the push for exascale and energy-efficient computing. Thus for the model to not currently support prediction of GPGPU performance is a significant limitation.

6.2 Future Work

This section provides an overview of future extensions to the work presented in this thesis. One strand of work is with the superscalar performance model, exploring improvements that may reduce outlier error and validating on more diverse architectures. The second strand of work is with the proxy-application MG-CFD, further validating MPI strong scaling, and identify optimisations for HYDRA.

6.2.1 Performance model improvements

In the design of the performance model, there was significant discussion around how to handle inter-instruction dependencies. To maintain simplicity in design, and avoid costly model training, it was decided to ignore all such dependencies. The resulting model provided generally good accuracy, but on the old architecture Westmere the errors of compute-bound predictions were high (20 % to 30 %). It would be very interesting to explore how an alternate strategy that explicitly models the dependencies performs, both in accuracy and in time to train. This work would use the recently released Open-Source Architecture Code Analyzer (OSACA) tool [34], intended as an open-source replacement of the end-of-life Intel tool IACA. This too models superscalar execution with port contention, but rather than estimate CPI values it uses pre-measured values, thus on its own may be unsuitable for novel architectures. It can identify the critical path of a sequence of instructions executing through a particular superscalar processor, using hard-coded CPI values. A visualisation is provided in Figure 6.1. Being open-source, their critical path detection could be ported into the model fitting process, greatly reducing development time; their use of pre-measured CPI values would be replaced with the model estimations.

The performance model also needs to be evaluated on more varied architectures, in particular the Fujitsu A64FX, and NVIDIA GPUs, that are architectures with high-bandwidth memory (HBM) well suited for HPC. Evaluating the A64FX should only require adding a new port model to the performance model, easily determined by official public documentation [35]. In all other respects, the A64FX can be modelled just like the Intel processors used in this thesis. NVIDIA GPU modelling has the potential to pose a challenge. One key difference to CPUs in regards to the modelling is what happens when an instruction requests data not in cache. In a CPU, the instruction will stall, and only ILP and hyperthreading can mask that stall. The cost of these stalls is captured in the estimation of CPI. In a NVIDIA GPU, which through its single instruction, multiple thread (SIMT) approach schedules 'warps' (i.e. batches) of work to CUDA cores, upon a data miss that entire warp

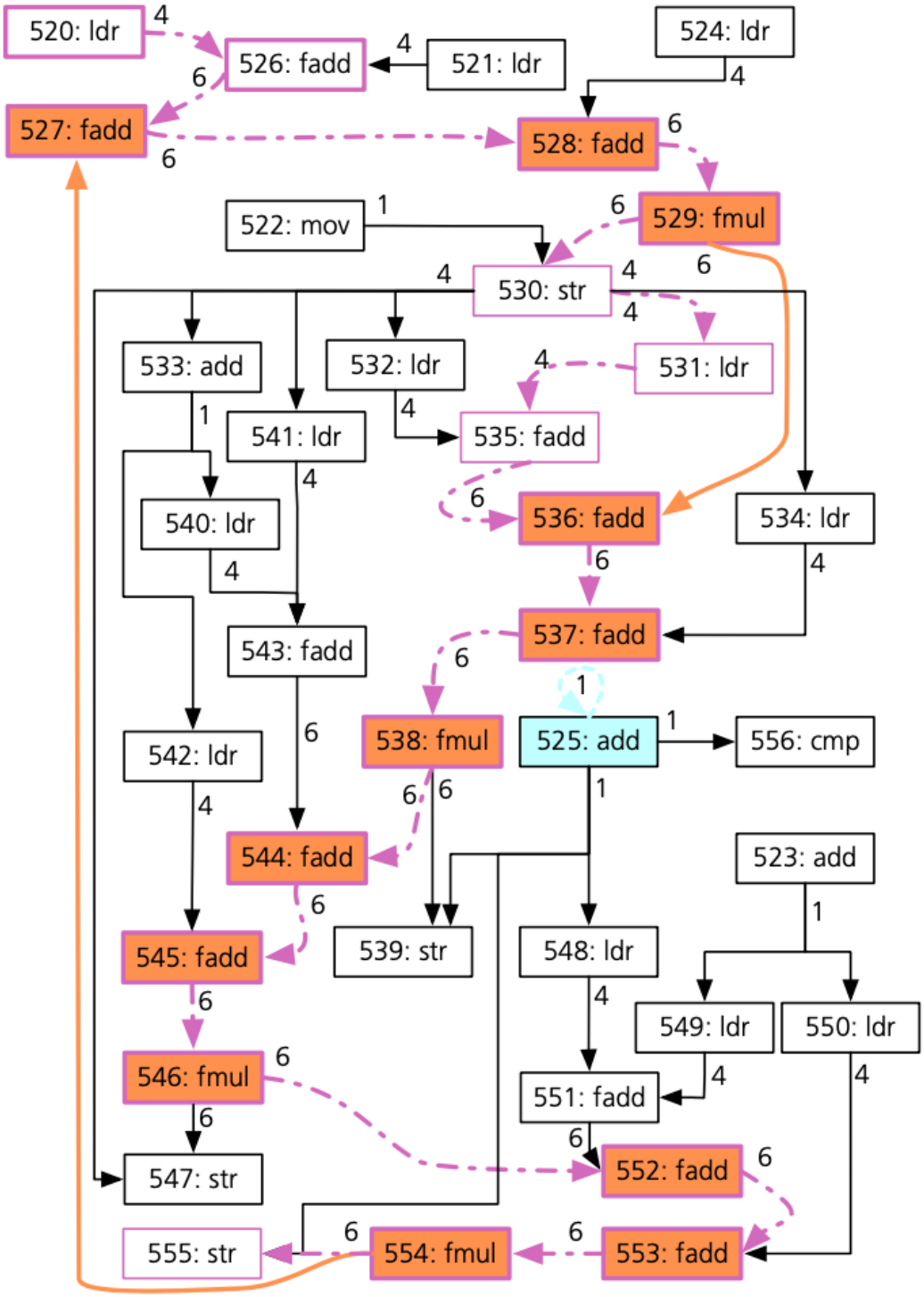


Figure 6.1: Critical path detection (pink) by OSACA tool, of Gause-Seidel loop on TX2 architecture [34]. Path numbers are instruction latency (CPI), in-box numbers are disassembly line numbers. Performance predicted to be bound mostly by floating-point execution.

is de-scheduled, and another warp scheduled for execution. It is possible that the current CPI estimation process can capture the cost of this rescheduling, thus not requiring further model development. The accuracy of the predictions will confirm if this is the case. If the error is high then the warp rescheduling will have to be explicitly modelled, but there is a plethora of CUDA GPGPU modelling research to draw from dating back to 2009 [32].

6.2.2 MG-CFD strong scaling optimisation

One of the improvements to MG-CFD in Chapter 4 is adding strong scaling functionality. This was validated to be very representative of HYDRA on a small cluster, and so should be validated further on a larger cluster. But as discussed then, due to the codes operating on the same mesh, and that this mesh through the partitioner determines load imbalance and communication pattern, then further validation is expected to succeed without issue. Thus thoughts can go immediately to exploring communication optimisation possibilities. One interesting avenue is considering carefully how the individual mesh partitions are allocated to processes, in terms of the physical cores on which they reside. With all HPC codes operating on multicore clusters, it is desirable to minimise the proportion of inter-process communication that occurs between different cluster nodes, and maximise communication between cores within the same node. There are two general approaches to this: (i) make the partitioner aware of the cluster topology, and (ii) reallocate partitions to ranks afterwards in a topology-aware manner. There is a variety of research on how to achieve this, but one paper in particular has investigated this specifically for unstructured CFD **??**. They model inter-node communication as costing 10× more than intra-node, then formulate the mapping problem as a quadratic assignment problem (QAP) seeking to minimise total communication cost. An NP-hard problem, they describe two low-cost heuristics that provide significant improvements to parallel efficiency.

Bibliography

[1] Ecp proxy apps suite. `https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/` (accessed March 13, 2019).

[2] Omar Aaziz, Jeanine Cook, Jonathan Cook, Tanner Juedeman, David Richards, and Courtenay Vaughan. A methodology for characterizing the correspondence between real and proxy applications. In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pages 190–200, 2018.

[3] Omar Aaziz, Jeanine Cook, Jonathan Cook, and Courtenay Vaughan. Exploring and quantifying how communication behaviors in proxies relate to real applications. In 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 12–22, 2018.

[4] R.F. Barrett, P.S. Crozier, D.W. Doerfler, M.A. Heroux, P.T. Lin, H.K. Thornquist, T.G. Trucano, and C.T. Vaughan. Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. Journal of Parallel and Distributed Computing, 75: 107–122, 2015.

[5] K Bergman, S Borkar, D Campbell, W Carlson, W Dally, M Denneau, P Franzon, W Harrod, K Hill, J Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 15, 2008.

[6] W. L. Briggs. Multigrid Tutorial. SIAM, Philadelphia, PA, 1987. ISBN 0898712211.

[7] S. Browne, C. Deane, G. Ho, and P. Mucci. Papi: A portable interface to hardware performance counters. Proceedings of Department of Defense HPCMP Users Group Conference, June 1999.

[8] R. A. Bunt. Performance Engineering Unstructured Mesh, Geometric Multigrid Codes. PhD thesis, The University of Warwick, September 2016.

[9] R. A. Bunt, S. J. Pennycook, S. A. Jarvis, L. Lapworth, and Y. K. Ho. Modelled optimisation of a geometric multigrid application. Proceedings of the 15th High Performance Computing and Communications (HPCC'13), pages 742–753, November 2013.

[10] R. A. Bunt, S. A. Wright, S. A. Jarvis, M. Street, and Y. K. Ho. Predictive evaluation of partitioning algorithms through runtime modelling. Proceedings of The 23rd IEEE International Conference on High Performance Computing, Data, and Analytics, pages 351–361, 2016.

[11] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A parallel framework for unstructured grid solvers. In Programming Environments for Massively Parallel Distributed Systems, pages 97–106. Birkhäuser Basel, 1994.

[12] P. E. Ceruzzi. A history of modern computing. MIT press, 2003. ISBN 0262532034.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. 2009 IEEE International Symposium on Workload Characterization (IISWC), pages 44–54, 2009.

[14] UK Mini-App Consortium. Uk mini-app consortium. `http://uk-mac.github.io/papers.html` (accessed March 6, 2016), 2016.

[15] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based cfd solvers on modern graphics hardware. International Journal for Numerical Methods in Fluids, 66(2):221–229, 2011.

[16] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, High Performance Computing, pages 489–507, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46079-6.

[17] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. IEEE Journal of Solid-State Circuits, 9(5):256–268, 1974.

[18] Roger Espasa, Mateo Valero, and James E Smith. Vector architectures: past, present and future. In Proceedings of the 12th international conference on Supercomputing, pages 425–432. ACM, 1998.

[19] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. ACM Transactions on Computer Systems, 27(2):3:1–3:37, May 2009.

[20] Massimiliano Fatica and Gregory Ruetsch. Chapter 1 - introduction. In Massimiliano Fatica and Gregory Ruetsch, editors, CUDA Fortran for Scientists and Engineers, pages 3–30. Morgan Kaufmann, Boston, 2014.

[21] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. Technical report, University of Denmark, 2018. [Online; accessed 17-July-2018].

[22] European Union Funding for Research and Innovation. Clean sky. `https://www.cleansky.eu`, 2019. [Online; accessed 21-June-2019].

[23] K. Fuji. Progress and future prospects of cfd in aerospace – wind tunnel and beyond. Progress in Aerospace Sciences, 41(6):455–470, 2005.

[24] Francis H. Harlow. Fluid dynamics in group t-3 los alamos national laboratory: (la-ur-03-3852). Journal of Computational Physics, 195(2):414–433, 2004. ISSN 0021-9991.

[25] M. Heroux and R. Barrett. Mantevo project. `https://mantevo.org/` (accessed March 3, 2016), March 2016.

[26] H. P. Hodson and R. G. Dominy. Three-dimensional flow in a low-pressure turbine cascade at its design condition. Journal of Turbomachinery, 109(2): 177–185, April 1987.

[27] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06), pages 114–122, 2006.

[28] Intel®. Intel® 64 and ia-32 architectures optimization reference manual. Technical report, Intel®, June 2016.

[29] Tanzima Z. Islam, Jayaraman J. Thiagarajan, Abhinav Bhatele, Martin Schulz, and Todd Gamblin. A machine learning framework for performance coverage analysis of proxy applications. In SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 538–549, 2016.

[30] F. T. Johnson, E. N. Tinoco, and N. J. Yu. Thirty years of development and application of cfd at boeing commercial airplanes, seattle. Computers & Fluids, 34(10):1115–1151, 2005.

[31] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium 2013 (IPDPS'13), pages 919–932, May 2013.

[32] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. A performance prediction model for the cuda gpgpu platform. In 2009 International Conference on High Performance Computing (HiPC), pages 463–472, 2009.

[33] L. Lapworth. Hydra-cfd: A framework for collaborative cfd development. Proceedings of the International Conference on Scientific and Engineering Computation 2004 (IC-SEC'04), June 2004.

[34] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. Automatic throughput and critical path analysis of x86 and arm assembly kernels. In 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 1–6, 2019. doi: 10.1109/PMBS49563.2019.00006.

[35] Fujitsu Limited. Fujitsu a64fx documentation. `https://github.com/fujitsu/A64FX`, 2019.

[36] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and A. J. Herdman. Experiences at scale with pgas versions of a hydrodynamics application. Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models 2014 (PGAS'14), pages 9–20, October 2014.

[37] L. Martinelli and A. Jameson. Validation of a multigrid method for the reynolds averaged equations. AIAA Journal, 1988.

[38] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In 1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425), pages 2–10, Oct 1999.

[39] P. Moinier, J. Müller, and M. B. Giles. Edge-based multigrid and preconditioning for hybrid grids. AIAA Journal, 40(10):1945–1953, October 2002.

[40] M. Molinari and W. N. Dawes. Review of evolution of compressor design process and future perspectives. Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 220(6):761–771, 2006.

[41] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PHJ Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In 2012 Innovative Parallel Computing (InPar), pages 1–12, May 2012.

[42] NASA. Tcgrid v. 400. `https://www.grc.nasa.gov/www/5810/rvc/tcgrid.htm` (accessed November, 8th 2017).

[43] A. Owenson. assembly loop extractor. `https://github.com/warwick-hpsc/assembly-loop-extractor` (accessed August 21, 2019), 2018.

[44] A. Owenson, S. Wright, R. Bunt, S. Jarvis, Y. Ho, and M. Street. Developing and using a geometric multigrid, unstructured grid mini-application to assess many-core architectures. In 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pages 68–76, 2018.

[45] A. Owenson, S. Wright, R. Bunt, Y. Ho, M. Street, and S. Jarvis. An unstructured cfd mini-application for the performance prediction of a production cfd code. Concurrency and Computation: Practice and Experience, 2019.

[46] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84), pages 348–354. ACM, 1984.

[47] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors. Proceedings of the 27th International Parallel and Distributed Processing Symposium 2013 (IPDPS'13), pages 1085–1097, May 2013.

[48] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Design and development of domain specific active libraries with proxy applications. Proceedings of Cluster Computing 2015 (CLUSTER'15), pages 738–745, September 2015.

[49] L. Reid and R. D. Moore. Design and overall performance of four highly loaded, high speed inlet stages for an advanced high-pressure-ratio core compressor. Technical report, NASA Lewis Research Center, Cleveland, OH, 1978.

[50] T. Röhl, J. Treibig, G. Hager, and G. Wellein. Overhead analysis of performance counter measurements. In 2014 43rd International Conference on Parallel Processing Workshops, pages 176–185, Sep 2014.

[51] Richard M. Russell. The cray-1 computer system. Commun. ACM, 21(1):63–72, January 1978.

[52] Shweta Salaria, Aleksandr Drozd, Artur Podobas, and Satoshi Matsuoka. Predicting performance using collaborative filtering. In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pages 504–514, 2018. doi: 10.1109/CLUSTER.2018.00066.

[53] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu. Performance projection of hpc applications using spec cfp2006 benchmarks. 2009 IEEE International Symposium on Parallel Distributed Processing, pages 1–12, May 2009.

[54] Sameh Sharkawi, Don DeSota, Raj Panda, Stephen Stevens, Valerie Taylor, and Xingfu Wu. Swapp: A framework for performance projections of hpc applications using benchmarks. In 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, pages 1722–1731, 2012.

[55] P. R. Spalart and S. R. Allmaras. A one-equation turbulence model for aerodynamic flows. AIAA Journal, pages 5–21, 1994.

[56] T. M. Taha and S. Wills. An instruction throughput model of superscalar processors. In 14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings., pages 156–163, June 2003.

[57] James E. Thornton. Parallel operation in the control data 6600. In Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems, AFIPS '64 (Fall, part II), page 33–40. Association for Computing Machinery, 1964.

[58] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development, 11(1):25–33, 1967.

[59] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. Proceedings of the Role of Reactor Physics Toward a Sustainable Future 2014 (PHYSOR'14), pages 1–12, September 2014.

[60] U. Trottenberg, C. W. Oosterlee, and A. Schuller. Multigrid. Elsevier, Amsterdam, The Netherlands, 2001. ISBN 978-0127010700.

[61] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. IEEE Transactions on Computers, 65(12):3537–3551, Dec 2016.

[62] P. P. Walatka, J. Clucas, R. K. McCabe, T. Plessel, and R. Potter. Fast user guide. Technical report, NASA Ames Research Center, 1994.

[63] David J. Wales and Jonathan P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. The Journal of Physical Chemistry A, 101(28):5111–5116, 1997.

[64] Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or cpu performance by analyzing public datasets. 15(4), January 2019.

[65] K. C. Yeager. The mips r10000 superscalar microprocessor. IEEE Micro, 16(2): 28–41, April 1996.